(10)

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: unclassified	AD-A173 122
1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT: unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	This document has been approved for public release and sale; its distribution is unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: The Regents of the University of California	6b. OFFICE SYMBOL (If applicable)
7a. NAME OF MONITORING ORGANIZATION: SPAWAR	
6c. ADDRESS (City, State, and ZIP Code): Berkeley, California 94720	7b. ADDRESS (City, State, and ZIP Code): Space and Naval Warfare Systems Command, Washington, DC 20363-5100
8a. NAME OF FUNDING / SPONSORING ORGANIZATION: DARPA	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code): 1400 Wilson Blvd., Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

* Parallel Unification Scheduling in Prolog

12. PERSONAL AUTHOR(S)

* Wayne Citrin

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
technical	FROM _____ TO _____	* September 18, 1986	* 64

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Enclosed in paper.

DTIC FILE COPY

DTIC ELECTE OCT 1 4 1986 S D E

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Productivity Engineering in the UNIX† Environment

Parallel Unification Scheduling in Prolog

Technical Report

S. L. Graham
Principal Investigator

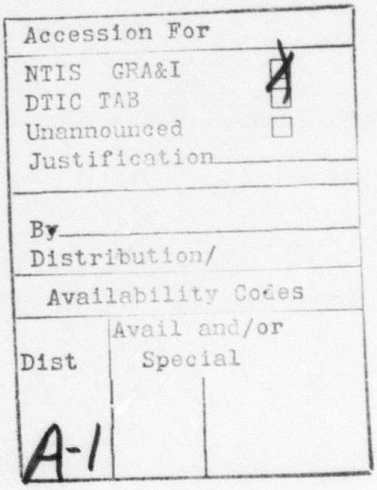
(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871



†UNIX is a trademark of AT&T Bell Laboratories

86 10 9 015

-A-

Parallel Unification Scheduling in Prolog

Wayne Citrin

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California

ABSTRACT

Unification, the fundamental operation in Prolog, can take up to 50% of the execution time of a typical Prolog system. One approach to speeding up the unification operation is to perform it on parallel hardware. Although it has been shown that, in general, there is no parallel algorithm for unification that is better than the best sequential algorithm, there is a substantial subset of unifications which may be done in parallel. Identifying these subsets involves gathering data using an extension of Chang's static data-dependency analysis (SDDA), then using that data to schedule the components of a unification for parallel unification. Improvements to the information gathered by SDDA may be achieved through procedure splitting, a source-level transformation of the program. This thesis describes and evaluates the above-mentioned techniques and their implementation. Results are compared to other techniques for speeding up unification. Ways in which these techniques may be applied to the Berkeley PLM machine are also described.

August 2, 1986

Table of Contents

Parallel Unification Scheduling in Prolog

Wayne Citrin

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California

1. Introduction

1.1. Prolog

Prolog is a logic programming language based on first order predicate calculus [18] . It is the implementation language being used in the Japanese Fifth Generation Computer Project [13] and the Berkeley PLM [9] and is used for a number of applications including expert systems and theorem proving. It has also been employed as an implementation language for compilers [4,25,26] .

It is expected that the reader is familiar with Prolog. For more information on the language, see [5] .

1.2. Unification

Unification is the fundamental operation in Prolog. It is the method by which Prolog variables are assigned values, and is also a test for equality. Unification is an operation in which two expressions are made identical by finding substitutions for some or all of the variables in the expressions. In Prolog, unification is performed when a procedure is called, the unification being attempted between the calling subgoal and the head of the clause being called. Unification may also be explicitly performed in Prolog by using the '=' operator.

In addition to controlling the assignment and comparison of values, unification affects the flow of control in Prolog programs. Failure of a unification may cause backtracking to occur, or may cause the next candidate clause in a program to be tried.

As an example of unification, to unify f(X,g(h)) and f(g(Z),g(Y)), where X, Y, and Z are variables, one possible solution would substitute g(Z) for X and h for Y. (Note that it is not necessary here to substitute anything for Z.) With these substitutions, both terms would become f(g(Z),g(h)). Note that not all pairs of expressions can be unified, nor is there always a unique unifier (substitution) for two terms which do unify. If two terms do unify, however, there is a **most general unifier** (mgu), which is unique up to renaming of variables [22] . If t and t' are two terms to be unified, and s is a unifier for t and t' (so that $s(t)=s(t')$), then s is the most general unifier of t and t' if and only if for any unifier p of t and t', there exists a substitution q such that $p=q \bigcirc s$ ($\bigcirc$ is the composition operation).

1.3. Motivation

In addition to being the fundamental operation in Prolog, unification of terms consumes about half of the execution time of a typical Prolog system. Woo [27] has found that a Prolog program executed by a UNSW Prolog interpreter running on a VAX 11/780 typically spends 55-70% of its total processing time performing unifications.

Even in the much more efficient Berkeley PLM, a similar amount of time is spent on unifications. In the PLM, there is a class of instructions, known as **get** and **unify** instructions, which perform unifications. [See section 9.1 for more information on these instructions.] In a set of six Prolog benchmarks, between 26% and 63% of the instructions executed by a PLM simulator were in this class. Similarly, measuring the percentage of microcycles spent executing these instructions (a measurement of the absolute time spent by the PLM on unification) yields comparable, although slightly lower, results. [See table 1.1.]

Table 1.1
Unification in the Berkeley PLM

Benchmark	% Get + Unify	% Get + Unify Microcycles
query	42.16	46.23
mu	26.52	44.49
serialize	63.26	44.98
deriv	60.37	23.37
nreverse	60.38	51.51
quicksort	56.06	37.74

1.4. Objectives

The evidence suggests that Prolog systems spend a large percentage of their time performing unifications. This reflects the ubiquitous nature of unification in Prolog programs in general. Reducing the time spent on unification would have a significant effect on the execution of Prolog programs. Woo [27] mentions two approaches to reducing the total time spent performing unifications in a Prolog program. The first approach is to reduce the number of unifications performed, either by transforming the Prolog program [14] or by selective backtracking [1,23] . The second approach, suggested by Woo, is to create more efficient sequential unification implementations, in his case by developing an efficient hardware unification unit.

There is a third approach, which is to reduce the amount of time spent performing unifications by performing at least some of the unifications simultaneously. When unifying two terms, several subterms must usually be unified. If some or all of these subterms could be unified simultaneously each time a unification had to be performed, a large speedup could result. This approach includes the solution proposed in this thesis, along with the dataflow scheme used in the Japanese PIM-D machine [16] , and a technique for transforming some unifications to term matching, a problem that can be solved quickly in parallel [19] . All of these other techniques will be addressed in chapter 8.

This dissertation will address the problem of how to speed up the unification operation and thereby reduce the percentage of program execution time spent on unification. The solution proposed is a compile time technique in which extensive preprocessing of a Prolog program is performed in order to determine which unifications may be scheduled at compile time for later parallel execution. Data for making these determinations is gathered using the static data-dependency analysis (SDDA) techniques originally developed by J-H Chang [2] , but in order to derive a satisfactory amount of parallelism in the unification, extensive refinements have been made. In addition, a source-level transformation technique called procedure splitting, which is driven by SDDA information, is used to increase the accuracy of this data still further. The information gathered will then be used to schedule the unifications. In addition, a similar run time technique will be discussed.

It should be noted that the three approaches mentioned above are not mutually exclusive, and particularly that some or all of them may be used in conjunction with the technique presented herein, which will be known as "parallel unification scheduling."

1.5. Other Parallelism

Prolog lends itself to other types of parallel execution [8] . Work has been done on the parallel execution of subgoals in a clause [2] (AND-parallelism), and on the parallel execution of candidate clauses [6,10] (OR-parallelism). These types of parallelism will not be addressed here, but there is no reason why they may not be included in a Prolog implementation along with parallel unification.

2. Unification - Theoretical Results

The fastest known sequential algorithm for unification was found by Paterson and Wegman [22] . It performs unification in time linear in the lengths of the two terms to be unified.*

Dwork [11] has shown that unification is log-space complete for FP.† Yasuura [28] has reached the same conclusion by a different method. This is not an encouraging result. It means that it is unlikely that a parallel unification algorithm could be significantly better than the best sequential algorithms. This is because if a problem is log-space complete for FP, then if the problem could be solved very quickly in parallel (say, in polynomial logarithmic time $\log^{o(1)}(n)$ using $n^{o(1)}$ processors, a class of problems known as NC), it would imply that NC = FP, that is to say that any problem solvable with a sequential polynomial algorithm could be solved very quickly in parallel. This is considered unlikely [7] . Thus, it is unlikely that any log-space complete problem can be solved much more quickly in parallel than sequentially.

It might be expected that Prolog head unification, that is, unification of a call subgoal and a clause head, would be a simpler case of the unification problem and thus possibly solvable quickly in parallel, since the two terms being unified may share no common variables before the unification begins, due to Prolog scoping rules. (For example, in general unification, if the terms f(X,a) and f(b,X) are to be unified, the two instances of X are considered to be the same variable, and substitutions for one must also be made for the other. In Prolog, however, if the former term is a call subgoal and the latter term is a clause head, Prolog scoping rules specify that the two instances of X represent different variables.) It is simple, however, to show that Prolog head unification is also log-space complete by construction a log-space, linear time reduction from general unification to Prolog unification:

Given two terms t1 and t2 (which may have common variables) to be unified, construct two new terms a(t1,t2) and a(X,X), where X is a variable that does not occur in t1 or t2. a(t1,t2) and a(X,X) have no common variables, so unifying them would be a Prolog head-type unification. It is obvious that t1 and t2 will unify (using general unification) if and only if a(t1,t2) and a(X,X) unify (using Prolog head-unification). This reduction can be performed in constant space and time. Since Prolog unification is a subset of general unification (i.e., any correct general unification algorithm will correctly unify two terms which do not share variables) and Paterson and Wegman [22] provide a polynomial (in this case, linear) algorithm for general unification, Prolog head unification is also in FP. This means that Prolog head unification is log-space complete, so it is also

---

* Paterson and Wegman provide a good example of their linear unification algorithm in operation.

† FP is the set of functions computable in polynomial time on a sequential processor. It is equivalent to the more well-known class P of decision problems decidable in polynomial time on a sequential machine. A problem, A, is log-space complete for a certain class (in this case, FP) if every problem in that class can be transformed to A using a transformation which takes space $O(\log(n))$ and time $O(n)$ where n is the size of the problem being transformed.

unlikely that a parallel algorithm to perform it would be significantly better than the best sequential algorithm.

It should be noted, however, that these results indicate that there is probably no algorithm which can unify any pair of terms in polynomial logarithmic time on a parallel machine. However, we can show that, although there are many term pairs that cannot be unified quickly on parallel hardware, there are many others which can. For example, in the next chapter, we will show that attempting to unify f(X,X) and f(a,b) is inherently sequential, while f(X,Y) and f(a,b) may be unified quickly on parallel hardware. There is evidence that most unifications in Prolog programs fall into this category (section 5.2 and [3]), so that it is profitable to unify Prolog terms on parallel hardware where possible.

3. Parallel Unification - An Overview

3.1. Assumptions

3.1.1. Language

The context of the unification operation is conventional sequential Prolog, for example the dialect of Edinburgh Prolog presented by Clocksin and Mellish [5] . The semantics of all operations, including unification, are assumed to be unchanged from this sequential version. There are a number of reasons for this. First, if the Prolog being used in this project is identical to conventional Prolog except for the parallelism in the unification operation, it will be possible to isolate the amount of improvement that is attributable solely to the parallel unifications. Additionally, simulation of conventional Prolog with parallel unification can be achieved by modifying an existing Prolog system rather than by building one from scratch. Details of the implementation of this simulator are given in section 10.4.

The second reason for assuming conventional sequential Prolog is that this is the language chosen for the Aquarius project [8] , into which it is hoped this work will be integrated. One of the objectives of the Aquarius project is to identify and exploit all available parallelism in a sequential Prolog program in order to achieve maximum performance. Schemes for doing this have been suggested by Chang [2] and Fagin [12] based on a simplified version of Conery's AND/OR execution model [6] and appropriate hardware has been suggested by Dobry [10] .

It should be noted that there are a number of other models for parallel execution of Prolog, which provide language extensions and semantic changes [24] . These models could incorporate the parallel unification scheme proposed here, although no one has yet attempted to apply static data-dependency analysis (SDDA) to these models. Even if SDDA and parallel unification scheduling were to be used in conjunction with these models, analysis of the effectiveness of the scheme would be complicated by the additional performance improvements from other parallel enhancements in the execution model.

3.1.2. Computational Model

The computational model to be used has been chosen for simplicity and efficiency. Intuitively, the subterms of one of the pair of terms to be unified are partitioned into a schedule (to be defined in chapter 4). The blocks of this schedule are ordered. For each schedule block, from first to last, a unification process is activated for each subterm in the block which unifies that subterm with the corresponding subterm in the other term being unified. These unification operations operate simultaneously and do not communicate with each other. They may be considered textual operations that modify the subterms being unified and all other variables in the terms that are affected by assignments to the variables in the two terms being unified. Considering these processes in textual terms avoids the complexities of parallel memory access. The result of two processes in the same schedule block assigning values to a variable, or assigning a value while others refer to it, is undetermined. When all the processes in a given schedule block complete, the processes in the next schedule block are activated.

Figure 3.1 shows an example of a schedule and its execution. (Note that the schedule in figure 3.1 is not an "optimal" schedule. but illustrates how a non-trivial schedule may have more than one step.)

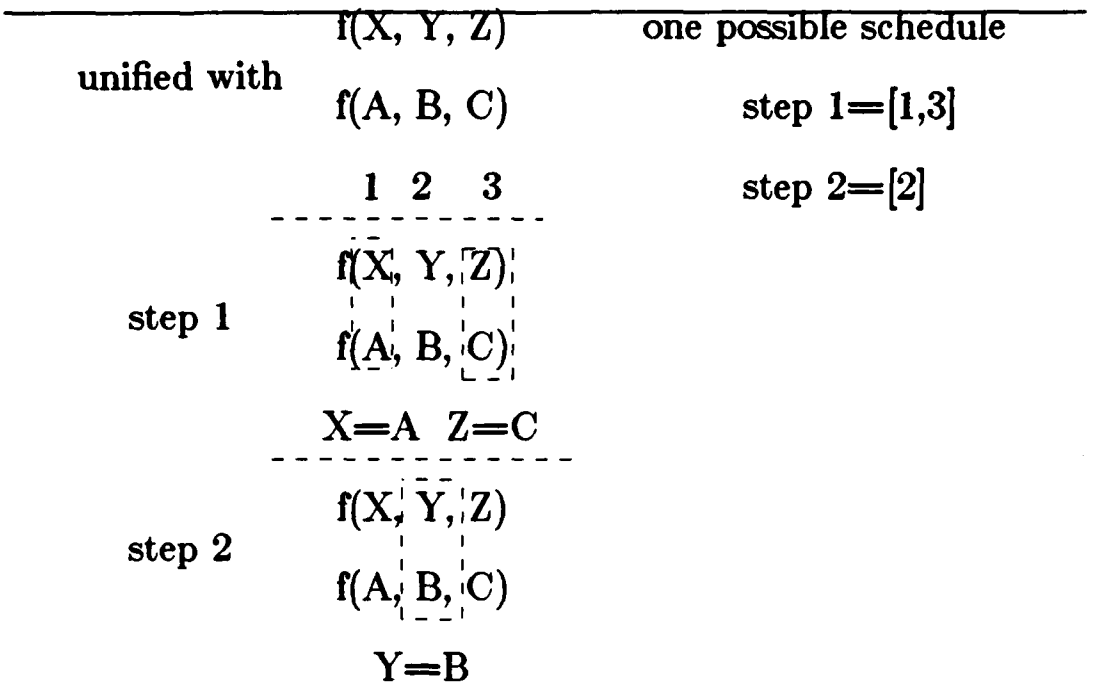


Figure 3.1 - A parallel unification

Since these unification processes are the only processes acting on the terms being unified, there is no mechanism to resolve conflicting simultaneous assignments to variables or to combine chains of unifications formed when a number of variables are bound together into equivalence classes. Figure 3.2 gives examples of both the above situations.

In figure 3.2(a), one unification process has unified X with a, the other with b*. The result is undetermined, whereas in reality the unification would fail, although here neither of the two processes have any knowledge of the actions of the other. In figure 3.2(b), one unification process has unified X and Y (rewriting both X and Y as some new common variable _1†); the other has unified X and Z (rewriting both X and Z as some new common variable _2). Such unifications are correct locally, but are incorrect in the global context of the two terms being unified. Which variable X has been unified to (_1 or _2) is undetermined, and

* By convention, upper-case letters in Prolog, like X and Y, represent variables, and lower-case letters, like a and b, represent constants.

† When two variables are unified, in order to avoid arbitrarily giving the unified variables one name or the other, we give the unified variables a new, unique name. We will use the Prolog convention of indicating generated variable names by an underscore followed by a number.

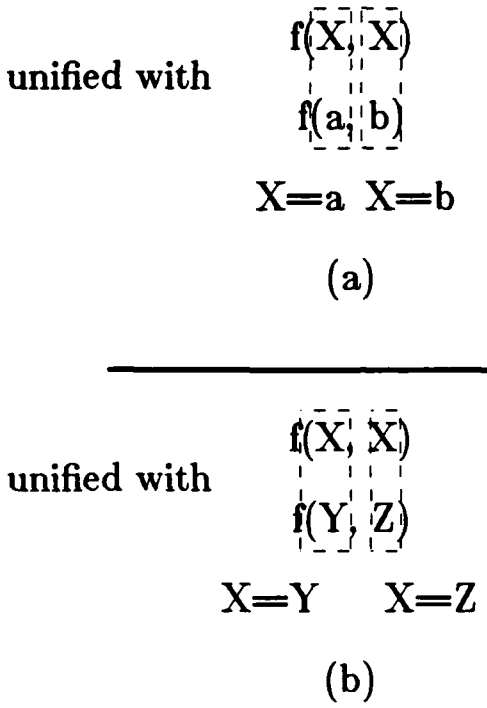


Figure 3.2
a) inconsistent bindings
b) incomplete bindings

there is no indication that Y and Z have been bound to each other as a result of this operation.

This model is computationally simple, requiring only identical unification processes and a sequencing mechanism. It can also be shown to map more easily onto modifications of existing Aquarius project hardware (chapter 9).

In constructing schedules for parallel unification of a subgoal and a clause head, we will always construct the schedule as a partition of the subterms of the clause head. This is because in Prolog, execution of head unification is associated with the called clause head and not with the called subgoal. In a call, control is transferred from the calling subgoal to the clause head, at which point both the call subgoal and clause head arguments have been seen and unification can take place. Parallel unification would require some mechanism to store subgoal arguments and execute the call, then a number of unification mechanisms, one for each unification proceeding in parallel. If the partition were associated with the call subgoal multiple call mechanisms would be required to store call arguments, locate clause head arguments, and perform unifications. Partitioning the clause head arguments is therefore much more efficient.

3.2. Intuitive notion of conditions for parallel unification

Intuitively, two terms can be unified in parallel if "they have nothing to do with each other." From the above assumptions, two terms may "have nothing to do with each other" if the terms contain no variables in common. Since one of the side effects of unification is to assign values to variables in the terms being unified, then if two pairs of terms being unified share no common variables, there is no possibility of two unification processes, one for each pair of unified terms, assigning values, possibly different, to the same variable. In such a situation, the two unification processes can safely go about their unification tasks without affecting the other pair of terms being unified.

If two pairs of terms being unified share a common variable, then two unification processes, each performing an independent unification, may derive incorrect results, as shown in the previous section. The problem is complicated by the fact that two term pairs may contain distinctly named variables which have been bound to each other or to a different common variable. An assignment to one of these variables will affect all the other variables to which it has been bound.

The determination of which variable pairs contain variables in common with, or bound to variables in, other terms cannot be done through a superficial examination of the text of the Prolog program. The execution of a program will assign values to variables so that at different points in the execution, some variables may be bound, and at other points they may be independent. Relationships between terms are also influenced by the values of data input to the program. J-H Chang has developed a technique, called static data-dependency analysis (SDDA) [2] which determines a worst-case bound on the relationships between variables in a Prolog program. Details of SDDA will be described in chapter 6. Here we will briefly describe the information yielded by SDDA.

SDDA classifies terms into three groups: ground terms, coupled terms, and independent terms.

Ground terms are terms which contain no unbound variables. They may either be constants (e.g., atoms or integers) or structures whose arguments are all ground terms. Ground terms may appear textually as variables; the point is that when execution of a program reaches the point in the text at which that term appears, it will be instantiated to a ground term.

A coupled term is one which shares a common variable with another coupled term, or which contains a variable which has been bound to a different variable in another term, and in which neither variable has been fully instantiated (i.e., made a ground term). Coupled terms may be partitioned into equivalence classes, known as coupling classes, containing all terms coupled together.

An independent term is a non-ground term which contains unbound variables which neither appear in any other term nor are coupled to variables in any other term.

When two terms are to be unified, SDDA may be used to partition the subterms of each term into classes of ground, coupled, and independent variables. The coupled subterms may be further partitioned into coupling classes.

These concepts will be more rigorously defined in later chapters. In the remaining sections of this chapter, a number of examples will be given to show how these concepts apply to unification.

3.3. A simple example

We are given a Prolog program containing the call subgoal f(X,Y) and the clause head f(a,b). Unifying the call subgoal and the clause head requires that corresponding subterms, in this case X and a, and Y and b, be unified. We would like to know whether X and a may be unified simultaneously and independently with Y and b. First assume that X and Y are independent terms, i.e., they contain no common or coupled variables. a and b are obviously ground terms and contain no common or coupled variables. We can be satisfied that the pair X and a and the pair Y and b "have nothing to do with each other" and can be unified simultaneously. As can be seen in figure 3.1, X is assigned the value a on unification, and simultaneously Y is assigned the value b. The subgoal and the clause head unify to f(a,b).

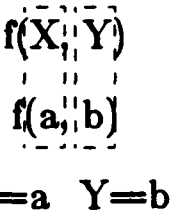


Figure 3.3 -
A correct parallel unification

Let us now assume that X and Y are coupled terms, in particular, that X and Y have been bound to each other. If we adhere to our initial assumptions concerning the computational model and do not wish to check that parallel unifications are consistent, simultaneously unifying X with a and Y with b will yield incorrect results, as shown in figure 3.4(a). Having independently assigned a to X and b to Y, there is no way to detect that X and Y may only be assigned one of these values and that the unification should fail. If the unifications are done sequentially (figure 3.4(b)), the process unifying Y and b already has the information that X (and Y) have already been assigned the value a and so the unification fails.

3.4. More complex examples

When two pairs of subterms to be unified both contain coupled subterms, but the coupled subterms are in different coupling classes, the two pairs still "have nothing to do with each other" and can be unified simultaneously. As an example, consider the unification of a subgoal f(A,B,B) and a clause head f(X,X,Y) where the variables A and B are independent, as are the variables X and Y. However, the second and third terms of f(A,B,B) are the common variable B, and so

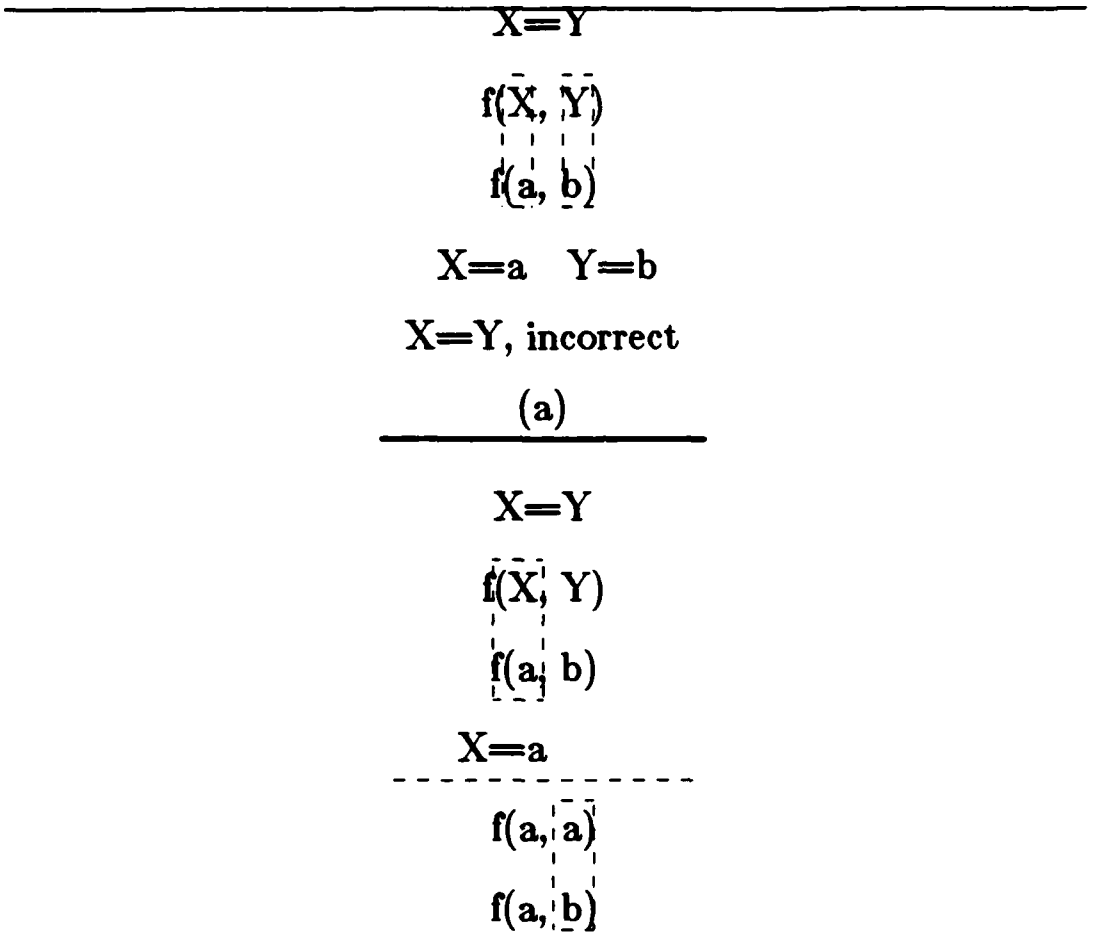


Figure 3.4 -
a) an incorrect parallel unification
b) a correct version of (a)

are coupled. Likewise, the first and second terms of f(X,X,Y) are the common variable X, and are also coupled. X and B, however, are independent of each other before unification according to the scoping rules of Prolog. As is shown in figure 3.5(a), we may simultaneously unify the first and third terms of the subgoal and clause head, and subsequently unify the second terms. Although the first and third terms contain coupled terms, which will ultimately be coupled to each other upon completion of the entire unification, during the first step they are in different coupling classes and therefore independent of each other. Unifying the second terms connects the coupling classes and completes the unification.

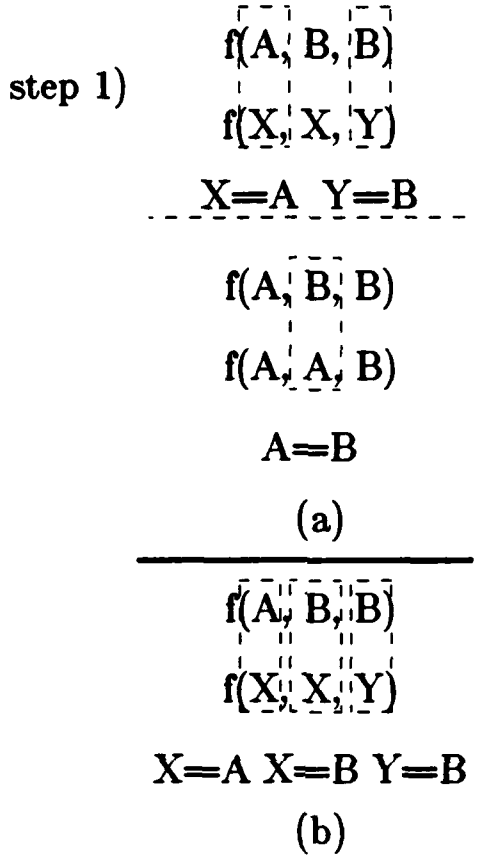


Figure 3.5 -
a) parallel unification involving two coupling classes
b) an incorrect version of (a)

Simultaneously unifying all three pairs of terms, as in figure 3.5(b) would violate the computational model assumption stated in section 3.1 that there is no way to gather the results of a number of simultaneous unification processes into equivalence classes. In this case, process 1 has unified X with A, and process 2 has unified X with B. Since the unification processes do not communicate, the implied unification of A and B, or of X and Y, is not performed.

The previous example showed that the two terms may be in different coupling classes at one point in the unification and in the same class at a subsequent point. It is also possible that a coupling class can be broken up as a result of some unifications. Consider the subgoal f(a,X,Y) to be unified with the clause head f(A,A,A). The first subterm of the subgoal is a ground term, and the second and third subterms will be assumed to be independent. In the clause head, all three subterms are the common variable A and are therefore coupled.

To simultaneously unify all three pairs of subterms again will yield indeterminate results according to computational model (figure 3.6(b)). However, if the first subterm pair is unified first, assigning the constant a to A, the second and third subterms of the clause head, previously coupled, now both become ground terms and may be simultaneously unified with their corresponding subterms in the subgoal, as shown in figure 3.6(a).

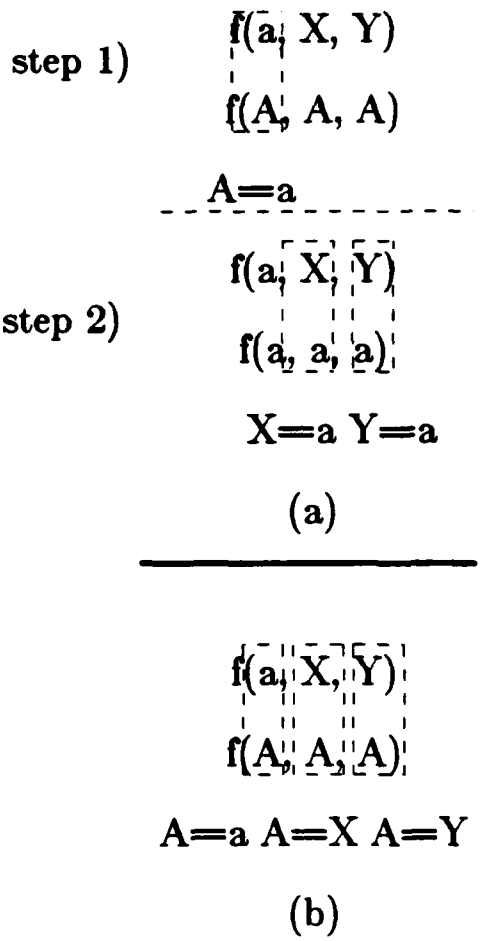


Figure 3.6 -
a) splitting coupling classes
b) an incorrect version of (a)

In the previous examples, structures have not been considered. The complexities in parallel unification scheduling involving structures will be discussed in chapter 4. One example showing one of the additional problems with structures will be presented here.

Consider the unification of the subgoal f(A) and the clause head f(g(X)) (figure 3.7). Recall the discussion of "atomic" units of unification in the previous section. The clause head has two atomic units: the functor g/1 and the variable

X which is the first argument of the structure whose functor is g. Note that the unification operation requires two operands. If both operands are not present, the unification operation cannot proceed. A unification can be performed between the variable A and the functor g/1, but no unification involving the variable X can be performed at that time since there is no term with which it may be unified. We say that X has no **corresponding subterm** in the subgoal at this time. After g/1 is unified with A, X does have a corresponding subterm (a dummy variable generated during the first unification) and may be unified.

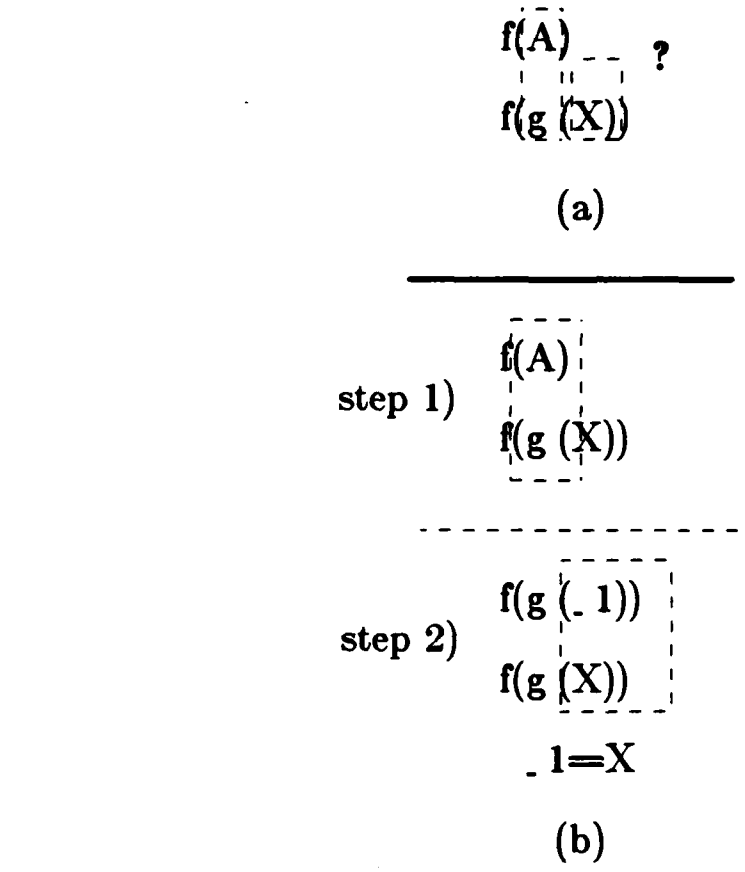


Figure 3.7 -
a) incorrect structure unification
b) correct structure unification

4. Parallel Unification Scheduling

To simplify understanding of the techniques involved in parallel unification scheduling, two versions will be presented. The first, simpler version describes parallel unification scheduling when it is assumed that structures do not appear. In this case, subterms may only appear textually as variables or constants, and the values assigned to variables may only be variables and constants. Using these assumptions, most of the principles behind parallel unification scheduling may be presented and understood.

Following that, a second, more complex version of parallel unification scheduling will be presented. In this version, subterms may be structures (whose arguments, too, may be structures), and likewise variables may contain structure values. Broadening the assumptions in this way requires considerable extensions to the first approach and requires considerably more precise SDDA information to derive an efficient schedule.

4.1. Without Structures

As mentioned above, in this section we will describe parallel scheduling for the unification of two terms, a clause head and a call subgoal, where the terms are of the form $f(t_1, \ldots, t_n)$ where the subterms $t_1, \ldots, t_n$ are either variables or constants at compile time. Likewise, we may assume that at the time of unification (at run time) the variables in the call subgoal will be either unbound, or bound to a constant or another variable. The variables in the clause head term will be assumed to be unbound, as they would be at the time of a call. The "atomic" unit of unification that is to be scheduled is the unification of a single subterm t_i of the clause head term, that is, unification of a variable or a constant, with the corresponding subterm of the calling subgoal.

4.1.1. Definitions

In the definitions below, assume a clause head $t=f(t_1, \ldots, t_n)$ and a call subgoal $t'=f(t_1', \ldots, t_n')$ which are to be unified. The subterms $t_1, \ldots, t_n, t_1', \ldots, t_n'$ are either variables or constants.

A **schedule** for the parallel unification of t and t' is a partition $\prod$ of the set of subterms of t, $\{ t_1, \ldots, t_n \}$, such that if t_i and t_j are in a block $\prod_k$ of $\prod$, then t_i and t_i' may be unified at the same time that t_j and t_j' are unified. Furthermore, the blocks are ordered, so that if $\prod_i$ and $\prod_j$ are blocks of $\prod$ and $i<j$, all elements of $\prod_i$ must be unified with their corresponding subterms in t' before any of the elements of $\prod_j$ are unified with their corresponding subterms. We can represent this ordering by the notation $\prod_i < \prod_j$.

A **mode**, similar to, but not the same as, the modes proposed by Mellish [20] , is a representation of the relationships between the subterms in the clause head and in the call subgoal. A mode is an n-tuple $(m_1, \ldots, m_n)$ where m_i describes the status of subterm t_i (in the case of a head mode), or t_i' (in the case of a goal mode). Each m_i has a value of g, c_j, or i, where g indicates that the subterm is a ground term (a constant or a variable which has been assigned a constant value), c_j indicates that the subterm is a coupled term, in this case a variable which has

been bound to other variables, but not to a constant (and where all the subterms to which it is coupled also have the mode c_j; there is a different value of j for each group of coupled variables), and i denotes an independent term, a term which is neither a ground term nor a coupled term. A mode that describes the relation between subterms in a clause head is a **head mode**, and a mode which describes the relation between subterms in a call subgoal is a **goal mode**.

A **current mode** C_i is a pair (G_i,H_i) describing the relationships between all subterms in the clause head and call subgoal being unified after all subterms in $\prod_1$ through $\prod_i$ are unified with their corresponding subterms in the call subgoal t'. G_i in the above pair is the goal mode and H_i is the head mode. C_0 is known as the **entry mode** and will be discussed in more detail below. It will be shown in the next section that C_i may be calculated from C_{i-1} and $\prod_i$. Thus, any C_i can be calculated from C_0 and $\prod_1$ through $\prod_i$. The actual text of t and t' are not necessary.

The entry mode C_0 is composed of the **goal entry mode** G_0 and the **head entry mode** H_0. G_0 is calculated using static data-dependency analysis as described in chapter 6. For the moment, we will simply assume that it exists and is available. It represents the coupling relationships between subterms of the call subgoal at the time of the call but before any unification with the clause head has begun.

H_0, the head entry mode, can be easily derived from the text of the clause head itself. This is because in Prolog the variables in a clause are unbound upon entry to that clause. Each invocation of a clause results in a new set of variables unrelated to any other set which may still be active. Every subterm which is a constant is of course a ground term. Any subterms which are commonly named variables are coupled terms in the same coupling class. Any subterm which is a variable that appears only once in the clause head is an independent term. Figure 4.1 shows some clause heads and their head entry modes.

Figure 4.1 - Head Entry Modes

clause head	H_0
f(a,b) :-	[g,g]
f(X,a) :-	[i,g]
f(X,X) :-	$[c_1,c_1]$
f(X,Y) :-	[i,i]
f(X,Y,X) :-	$[c_1,i,c_1]$
f(X,Y,Y,X) :-	$[c_1,c_2,c_2,c_1]$

4.1.2. Determining Schedule Safety

We now define the notion of a safe schedule for parallel unification of two terms by providing an algorithm for determining a schedule's safety. Such an algorithm could conceivably be used to create safe schedules, but it would be inefficient. The algorithm presented here is simply used to define the condition.

The key to the safety of a schedule is that, with one exception, for any pair of subterms in a schedule block, none of the four subterms involved (the two

subterms in the clause head and their corresponding subterms in the call subgoal) be coupled to each other. The one exception is that a subterm in a clause head and its corresponding subterm in the call subgoal may be coupled. In fact, unifying the coupled subterms is essentially a no-op, and such a subterm may be discarded from the schedule. Figure 4.2 gives a simple example of this. In the schedule given, subterm 1, the first X, is unified with its corresponding subterm, the first A. Now, prior to the second block, which contains the second subterm, this subterm and its correspondent in the subgoal are coupled. It is not necessary to perform the unification. We will see, however, that when SDDA yields the information that two terms are coupled, it really means that it is possible that they may be coupled when execution reached that point in the program. In other possible cases, the two terms may be independent. However, if we know absolutely that the two terms are coupled any time the clause is called, we may use this optimization.

Briefly, the reason that two coupled terms need not be unified is that the condition of coupling implies that the two terms are either identically named unbound variables, have already been unified with each other, or have been unified with a common variable. Thus, another unification would be redundant.

As mentioned earlier, the schedule is associated with the clause head, rather than the subgoal, or with the clause's procedure as a whole. There are a number of reasons for this. The first reason is that each procedure may have several clauses. Each clause will have its own head entry mode, H_0, which can easily be computed. The H_0's will likely be different for each clause in the procedure and such differences may require different schedules. The schedule is not associated with the subgoal because, first, in Prolog execution, unification cannot begin until both the subgoal **and** the clause head have been seen, i.e., until control has passed from the subgoal to the clause head. Secondly, the same Prolog procedures may be called in many different ways from many different subgoals each of which may have a different goal entry mode and which may even have goal entry modes that vary at different places in the execution of the program. In fact, in the static case (section 5.2), goal entry modes used for unification scheduling are actually generalizations of all entry modes which may occur during a call to that clause, combined to form a worst-case estimate. This is because a clause may be called from a number of sites in a program, each of which may call the clause with a different mode. It is even possible that a clause may be called several different ways from a single calling site. This will be explained further in chapter 6, but we may assume that this worst-case estimate is actually the precise goal mode of a single call subgoal which is the only site from which the clause in question is called. It can be seen, however, that the only constant in the above candidates for association with a schedule is the clause head, with its unvarying head entry mode, and thus the best entity with which to associate the schedule. Also, and not least important, it will be seen that such a scheme maps well onto the proposed implementation described in chapter 9.

Algorithm 4.1 gives a decision procedure for determining the safety of a schedule.

Figure 4.2: Unifying Coupled Variables		
subgoal	...,f(A,A),...	G_0=(c_1,c_1)
clause head	f(X,X) :-	H_0=(c_2,c_2)
	(1) (2)	
		$\prod_1 = \{(1)\}$
After $\prod_1$:		
	...,f(_1,_1),...	G_1=(c_1,c_1)
	f(_1,_1) :-	H_1=(c_1,c_1)
		$\prod_2 = \{(2)\}$
$\prod_2$ unnecessary		

Algorithm 4.1: decision procedure for schedule safety.

Input:

A clause head $t = f(t_1, \ldots, t_n)$ and a call subgoal $t' = f(t_1', \ldots, t_n')$.

$C_0 = (H_0, G_0)$ containing:

A goal entry mode G_0 computed by SDDA.

A head entry mode H_0 computed from t.

A schedule $\prod = \{\prod_1, \ldots, \prod_m\}$ for parallel unification of t and t'.

Output:

SAFE if schedule is safe, UNSAFE otherwise.

Algorithm:

For each schedule block $\prod_i$ from $\prod_1$ to $\prod_n$:

1: if there exist two terms $t_j, t_k \in \prod_i$ such that at least one of the pairs $(G_{i-1,j}, G_{i-1,k})$, $(G_{i-1,j}, H_{i-1,k})$, $(H_{i-1,j}, G_{i-1,k})$, $(H_{i-1,j}, H_{i-1,k})$ [$H_{i,j}$ is the element in H_i corresponding to t_j, similarly for $G_{i,j}$.] is (c_l, c_l) for some c_l, output 'UNSAFE' and halt.
else compute C_i according to the function C_i = next_C(C_{i-1},$\prod_i$).

output 'SAFE' and halt.

Function next_C(C_{i-1},$\prod_i$) is computed as follows:

$(H_i, G_i) = (H_{i-1}, G_{i-1})$.

for j = 1 to n

if $t_j \in \prod_i$

(H_i, G_i) = table(H_i,G_i,j) according to the table 4.1 below.

If the mode for a coupling class c_k or c_l is **augmented** (see section 6), i.e., if we know that all subterms in that coupling class are always coupled regardless of the way that clause is called in the program, we may modify the table as shown in table 4.2.

If it is known that a term is always coupled to other terms at a certain point in a unification schedule, then if one of the coupled terms is unified with a ground

Table 4.1 unification simulation table.

$H_{i-1,j}$ \ $G_{i-1,j}$	g	i	c_k
g	a	a	a
i	a	b	c
c_l	a	d	e

a) $H_{ij}=G_{ij}=g$.

b) $H_{ij}=G_{ij}=c_m$ for some new m.

c) $H_{ij}=G_{ij}=c_k$.

d) $H_{ij}=G_{ij}=c_l$.

e) all elements of H_{i-1} and G_{i-1} which are c_k or c_l are replaced in H_i and G_i with c_m for some new m.

Table 4.2 augmented unification simulation table.

$H_{i-1,j}$ \ $G_{i-1,j}$	g	i	c_k
g	*	*	a
i	*	*	*
c_l	b	*	*

*) unchanged from table 4.1

a) $H_{ij}=g$, and all elements of H_{i-1} and G_{i-1} which are c_k are replaced in H_i and G_i with g.

b) $G_{ij}=g$, and all elements of H_{i-1} and G_{i-1} which are c_l are replaced in H_i and G_i with g.

term, all of the coupled terms become ground terms. This cannot be done without augmented information since terms in a non-augmented coupling class may or may not actually be coupled. As is shown in chapter 6, in the absence of augmentation, we must include potentially coupled terms in a coupling class. Since augmentation allows us to eliminate entire coupling classes, it allows greater opportunities for parallelism in the schedule. After unifying one element of a coupling class with a ground term, all the other coupled terms become ground and can be unified in parallel (assuming that the corresponding terms are not coupled). Thus, it will be necessary to develop techniques for augmentation to take advantage of substantial additional opportunities for parallelism; these techniques will be explained in section 6.2.

4.1.3. Scheduling Algorithm

4.1.3.1. Algorithm

The proof that unification scheduling is NP-complete requires a number of ideas introduced in this and following sections, and so will be deferred to the end of the chapter. However, given that the problem is NP-complete, there are a number of approaches that may be taken. One is to choose at random a schedule, test it according to the polynomial algorithm of section 4.1.2, and keep trying schedules until a safe one is found which has a sufficiently small number of parallel steps. Where the number of unifications to schedule is small, this may be a feasible approach.

Another approach is to design an efficient algorithm which provides good (optimal or near-optimal) results in most cases. Three such heuristic algorithms will be presented here.

Two of the algorithms to be presented are "local" heuristic algorithms. Given a set of unifications which have not yet been scheduled, the algorithms select the unifications to be scheduled in the next schedule block. The process is repeated until all unifications are scheduled. The two algorithms differ in the method by which unifications are selected for scheduling. The third algorithm is a "global" heuristic algorithm, in which the entire schedule is taken into account. In this algorithm, an initial schedule is provided and then repeatedly improved by migrating unifications upwards (that is, from the end toward the beginning of the schedule). The process may be repeated as often as desired or until no further improvements are made in the schedule. All three algorithms have been implemented, and a comparison of their performance is given in chapter 11.

In the global algorithm, a simple initial schedule is provided, then repeatedly improved upon. This schedule may start with one unification per schedule partition, ordered simply as the unifications appear from left to right in the original program text, or perhaps ordered so that independent pairs, type-1 pairs, type-2 pairs, and type-3 pairs appear in that order. (These terms are defined later, with the third algorithm.)

The object of the algorithm is, starting from the end of the schedule, to move unifications to the front of the schedule, adding them to earlier schedule blocks. This has the effect of increasing the parallelism in earlier blocks, and, hopefully, emptying later blocks so that they disappear, thereby shortening the effective length of the set of unifications. Moving unifications from the end forward, rather than the other direction, was chosen because it is necessary to recompute all modes from the block to which the schedule was moved until the end of the schedule. The direction of migration was chosen to minimize the number of node recomputations.

Starting from the end of the schedule, a unification is selected and an attempt is made to move it to an earlier block. The first block found to which the unification may be moved (the search is made from back to front), that is, where the candidate unification is not coupled to any other element of the block, is the block to which the unification is moved. It might be possible to move the unification even farther forward; if so, this will be discovered on a subsequent

iteration. After the unification has been moved, all subsequent modes are recomputed. If any subsequent block has been made unsafe by the movement, the unification is moved back to where it was found. If the original search yields no block into which the unification could be moved, that is, the search reached the first block without finding a safe destination, the unification is not moved. This process is repeated until movement has been attempted for all unifications in all blocks. The entire process can be repeated as many times as desired, or until no further improvement is yielded. Algorithm 4.2 gives this global heuristic scheduling algorithm.

It might be interesting to formulate this algorithm as a rewriting system according to the Knuth-Bendix scheme [17] , which it somewhat resembles. A number of problems would have to be overcome, however. First, since there may be more than one solution, it may be possible that no set of axioms implementing a scheduling system will be complete (i.e., derives exactly one irreducible schedule from a given input schedule). Second, since the scheduling problem is NP-complete, a Knuth-Bendix derivation will probably take exponential time to execute, while the similar heuristic algorithm presented here will work in polynomial time. It might be possible to limit the number of reductions in any derivation to fall within a polynomial bound, which would usually leave the derivation incomplete, and it is not clear how quickly the Knuth-Bendix method converges on an optimal schedule. The above algorithm, on the other hand, is guaranteed to reach a stable solution within a polynomial bound. Finally, the axioms/lemmas required for transformations are substantially more complex than any of the toy group theory problems presented in Knuth and Bendix's paper.

Algorithm 4.2 - Global heuristic scheduling algorithm

Input:
$C_0=(G_0,H_0)$
$t=f(t_1,\ldots,t_n)$

Output:
$\Pi=\{\Pi_1,\ldots,\Pi_m\}$

Algorithm:
given the set {1,...,n}, partition the set into
a schedule Π so that each block Π_i
is a singleton set.
compute $C_1,\ldots,C_n$ using the next_C function.
p_size = n;
foreach i from p_size to 2
foreach $p\in\Pi_i$
foreach j from $i-1$ to 1
if p can be safely placed in Π_i
/* if $G_{j-1,p}=c_k$ or $H_{j-1,p}=c_k$, for some k,
and there is no $q\in\Pi_j$ such that
$G_{j-1,q}=c_k$ or $H_{j-1,q}=c_k$ */
then
move p from Π_i to Π_j
recompute $C_j,\ldots,C_{p\_size}$ using next_C
if there exists $l\in j,\ldots,p\_size$ such that
Π_l is unsafe according to C_{l-1}
/* if there exists a $q,r\in\Pi_l(q\neq r)$
such that $G_{l-1,q}=c_k$ or $H_{l-1,q}=c_k$ (for some k)
and $G_{l-1,r}=c_k$ or $H_{l-1,r}=c_k$ */
then move p back to Π_i and break loop
if $\Pi_i=\{\}$
then
p_size = p_size - 1
foreach k from i to p_size
$\Pi_k=\Pi_{k+1}$

As mentioned before, there are many ways by which the original order may be chosen. The left-to-right approach is basically a random ordering. The classification by type (independent, type-1, -2, -3) may be more successful. The description of the second local algorithm will define the above types and provide the rationale for this ordering.

The remaining two algorithms are "local" algorithms, in that they are concerned with scheduling a single schedule block from the set of schedulable unifications and the current mode. The first approach is to simply schedule as

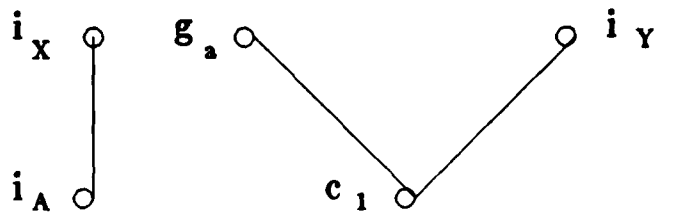
many unifications as possible. The second approach is to predict the influence of each unification on the parallelism in succeeding unifications and rank them according to that criterion. Then, as many "good" unifications as possible are scheduled, before less good unifications are considered. This approach may not yield as large a block as possible, but it may uncover additional parallelism for subsequent blocks.

The first of the local algorithms may be implemented by constructing a **coupling graph** from the current mode and the set of unscheduled unifications, in which each ground and independent term is represented by a node, as is each coupling class. Edges representing the correspondence of subterms in the two terms being being unified, and therefore unifications to be scheduled, are added to connect nodes associated with corresponding terms. [See figure 4.3(a) for an example of a coupling graph.] In a coupling graph, two unifications involving the same coupling class are represented by edges incident on the same node. Choosing a safe schedule block is equivalent to choosing a set of edges without common nodes. Choosing the largest safe schedule block is equivalent to choosing the largest set of edges with no common nodes [figure 4.3(b)]. This is the maximal matching problem, which can be solved in polynomial time. Since there are unifications leading to graphs which are not bipartite (see figure 4.3(c)), the problem can be solved in time $O(V^4)$ where V is the number of nodes in the graph [21] . In terms of unification, $V=G+I+C$ where G is the number of ground terms, I is the number of independent terms, and C is the number of coupling classes. The scheduling algorithm for parallel execution repeatedly constructs a coupling graph, finds the maximal matching, then alters the coupling status to reflect the newly scheduled unifications (according to the function next_C in table 4.2), repeating the process until all unifications are scheduled. If N were the total number of subterms in the two terms being unified, then the upper limit to the number of iterations would be $N/2$, the number of pairs to be scheduled (since in some cases, one unification per schedule block might be necessary, as in figure 4.3(c)). $N \geq G+I+2C$ (since each coupling group must be represented at least twice in a unification, otherwise the variable would be merely independent). The above algorithm performs scheduling in time $O(N*V^4)$, and since $N \geq V$, the time of execution is at least $O(V^5)$.

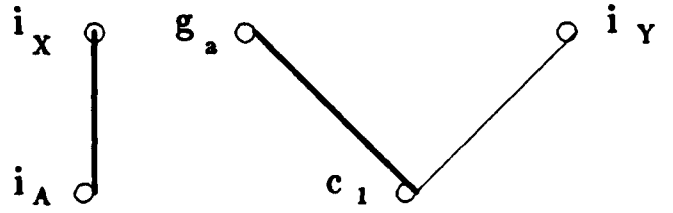
The second "local" approach improves on the first in a number of ways. First, it would be desirable to reduce the exponent in the time complexity from five to two or three. Secondly, although each iteration produces the maximal matching at that step, the particular choices of edges (and therefore of scheduled unifications) may reduce the size of maximal matchings in subsequent steps. For example, if a unification is scheduled which unified variables in two distinct coupling classes, in subsequent scheduling steps variables in the two classes may not be unified simultaneously, since the initial unification combined the two previous coupling classes into one. Had the initial unification not been done at that time, further parallel unifications involving variables in the two coupling classes could have been scheduled. Conversely, if a variable in a coupling class were to be unified with a ground term, all variables coupled to it would also automatically be unified with that ground term, and the coupling class would disappear. There would then be nothing to keep the formerly coupled variables, now ground terms,

a)

f(X,a,Y) $G_0 = (i,g,i)$
f(A,B,B) H_0 - (i,c_1,c_1)



b)



c)

G_i - (c_1,c_2,c_2)
$H_i = (c_3,c_3,c_1)$

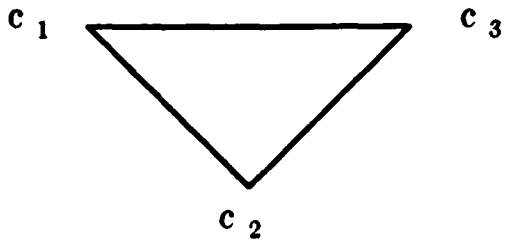


Figure 4.3
a) a coupling graph
b) a maximal matching for (a)
c) a non-bipartite coupling graph

from being scheduled in the same block in a subsequent step.

The heuristics to be used are, first, to delay combining coupling classes by unifying pairs of coupled variables until it is unavoidable, and second, to unify coupled variables with ground terms as early as possible in order to eliminate entire coupling classes and exploit more parallelism.

In addition, unification pairs composed only of ground and independent terms (having no coupled variables) may always be scheduled. These pairs, which we will call **independent pairs** may be discovered by simple inspection and need not be included in the coupling graph.

The algorithm for scheduling a step first schedules all independent pairs. It then sorts each of the remaining pairs (which each contain at least one coupled variable) into one of $3C$ bins, where C is the number of coupling classes. For each coupling class, there are three bins, one each for type-1, type-2, and type-3 pairs. The pairs are divided up into these groups as follows.

- Type-1 pairs contain a ground term. Unifying one of these pairs would cause all variables coupled to the variable in the pair to become ground terms in subsequent scheduling blocks (assuming augmented coupling information). This increases the available parallelism in these later blocks, since scheduling of those pairs will no longer be constrained by the previously existing coupling class.
- Type-2 pairs are those pairs that contain either an independent term or contain two terms which are members of the same coupling class. In the former case, the unification of such a pair will add the independent term to the coupling class, but since an independent term only appears once, it will not be seen again in subsequent scheduling blocks. In the latter case, the unification will not change the coupling configuration at all, since the terms are already coupled. In either case, the effect on subsequent schedule blocks is neutral; it neither increases nor decreases subsequent available parallelism.
- Type-3 pairs contain two coupled terms which are members of different coupling classes. When a pair of this type is unified, the two coupling classes are joined in subsequent scheduling blocks. This decreases the amount of available parallelism in subsequent blocks, since pairs which previously involved different coupling classes and could be scheduled to be executed in parallel may now involve the same coupling class and must be scheduled to be unified sequentially.

Once the pairs have been sorted, one pair is chosen from each of the type-1 bins. If a type-1 bin is empty, a pair is chosen from that coupling class' type-2 bin. A type-3 pair only becomes a candidate for scheduling if, in both coupling classes with which it is associated, the type-1 and type-2 bins are both empty. From all such candidate pairs, a coupling graph is created, and the maximal matching is found using the standard algorithm (again, for non-bipartite graphs). This matching is translated back into pairs and those pairs are scheduled.

Algorithm 4.3 provides the single-step scheduling algorithm.

Algorithm 4.3: Scheduling a single step - no structures.

Input:

$C_{i-1} = (G_{i-1}, H_{i-1})$
List S of pairs not yet scheduled.

Output:

$\prod_i$ (new schedule block).
new S.

Algorithm:

```
Π_i={};

/* initialize bins */
for k = 1 to C   /* C = number of coupling classes */
      for l = 1 to 3
            B_{k,l}={};

foreach j∈S
      /* schedule independent pairs */
      1: if j is independent pair
      then Π_i=Π_i∪{j};
      else begin /* sort remaining pairs */
            if j is type-1
            with coupled variable in c_k
            then B_{k,1}=B_{k,1}∪{j};
            else if j is type-2
            with coupled variable in c_k
            then B_{k,2}=B_{k,2}∪{j};
            else if j is type-3
            with coupled variable in c_k,c_l
            then begin
                  B_{k,3}=B_{k,3}∪{j};
                  B_{l,3}=B_{l,3}∪{j};
            end
      end

/* schedule type-1 and type-2 pairs */
2: for i= 1 to C
      if B_{i,1}≠{}
      then begin
            choose j∈B_{i,1};
            Π_i=Π_i∪{j};
      end
      else if B_{i,2}≠{}
```

```
        then begin
                choose j∈B_{i,2};
                Π_i=Π_i∪{j};
        end

/* find candidate type-3 pairs */
foreach (i,k)∈(1, . . . ,C)×(1, . . . ,C)
such that B_{i,1},B_{i,2},B_{k,1},B_{k,2}={}
and there exists j
such that j∈B_{i,3} and j∈B_{k,3}
        add j to coupling graph, G;

find maximal matching, M, of G;

/* translate matching to schedule */
3: for all (c_i,c_k)∈M
        begin
                find j∈B_{i,3}∩B_{k,3};
                Π_i=Π_i∪{j};
        end

/* generate new S */
S=S−Π_j;
```

The entire scheduling algorithm simply iterates over the single-step scheduling algorithm until all subterm unifications have been scheduled. The next_C function mentioned in algorithm 4.4 is the one initially mentioned in algorithm 4.1 and table 4.2. This function computes C_i from C_{i-1} and $\prod_i$ by simulating unification at the coupling status level. The single-step scheduling algorithm is represented in algorithm 4.4 by the function single_step. Both algorithm 4.3 and the first local algorithm may be used as the single_step function.

As mentioned before, all three scheduling algorithms, or rather their extensions that handle structures, have been implemented and tested. The results are given in chapter 11.

4.1.3.2. Analysis of Scheduling Algorithms

By inspection, it can be seen that the inner loop of the first scheduling algorithm (4.2) is executed $O(n^3)$ times for each execution of the algorithm, if n is the number of subterms to be scheduled. The outer loop is executed p_size times, where p_size is at most n. For each iteration of the outer loop, the second loop is executed once for each pair in the schedule block being examined. This can be at most n-1 (if it were n, the schedule would be a single block and the algorithm would have terminated). For each iteration of the second loop, the inner loop is executed at most n-1 times. Thus the $O(n^3)$ execution time. It may be necessary to repeat the entire process no more than n times before no further improvements can be made. Thus, the entire scheduling process will take $O(n^4)$ time.

Algorithm 4.4: Heuristic scheduling algorithm, no structures.

Input:

$C_0=(G_0,H_0)$,
$t=f(t_1,\ldots,t_n)$

Output:

$\Pi=(\Pi_1,\ldots,\Pi_m)$

Algorithm:

```
S={1, . . . ,n};
i = 1;
for S≠{}
    begin
        (Π_i,S)=single_step(C_{i-1},S);
        C_i=next_C(C_{i-1},Π_i);
    end
```

As mentioned in the previous section, the second scheduling algorithm will take $O(n^5)$ time.

The third unification scheduling algorithm has an execution scheduling time of $O(N^*\max(N,C^4))$, where N is the number of subterms in the clause head t, and C is the number of coupling classes present in the clause head and the call subgoal, as counted in the entry mode C_0.

The single_step subroutine (algorithm 4.3) can be divided into six parts. The first part, which initializes the bins into which type-1, 2, and 3 pairs are sorted takes time $O(C)$. The second part, sorting the pairs into independent pairs and bins for coupled pairs, takes time $O(N)$. A single iteration of the loop can be done in constant time, since recognition of the type of pair can be done in constant time, as can the depositing of the pair in a bin. Since the loop acts on each element in the set S, which can be of maximum size N, N iterations are performed, and the step can be completed in time $O(N)$.

The next step, in which type-1 and type-2 pairs are scheduled, takes $O(C)$, since C iterations of the loop are performed, and each iteration, which either determines that a bin is empty or selects a member of that bin, is done in constant time.

The location of candidate type-3 pairs can be done in time $O(P_3)$, where P_3 is the number of type-3 pairs. For each type-3 pair, the type-1 and type-2 bins for the corresponding coupling classes can be inspected in constant time. Edges in the coupling graph can be added in constant time if the graph is represented by an adjacency matrix.

The maximal matching of the graph G can be found in time $O(C^4)$ using the algorithm in [21] .

to translate the matching to a set of pairs to be scheduled, each edge in the matching must be matched with its corresponding pair. If the information relating edges to pairs was stored in the adjacency graph, each translation can be done in constant time. since the size of the matching M is bounded above by P_3, the number of type-3 pairs, the entire translation can be accomplished in P_3 iterations of the loop, so that this part takes time $O(P_3)$.

Since $P_3 \leq N$, one execution of algorithm 4.3 takes time $O(\max(N,C^4))$.

In considering the entire scheduling algorithm (4.4), the next_C subroutine (from algorithm 4.1) takes $O(N)$ time, since it can be implemented by going down the two tuples G_i and H_i, and modifying corresponding elements $G_{i,j}$ and $H_{i,j}$ if $j \in \prod_i$, according to table 4.2. If table 4.2 indicates that a coupling class must be changed to ground terms, or that two terms must be joined, the relevant information may be stored so that if such coupled terms are encountered later in G_i and H_i, the appropriate changes may be made.

Since, in the worst case, the main loop in algorithm 4.4 will require N iterations, one for every element of S, the entire algorithm will take time $O(\max(N,C^4))$.

4.1.3.3. Proof of Correctness

In this section, we briefly prove a theorem concerning the correctness of the third scheduling algorithm. In particular, we will show that the scheduling algorithm only generates safe schedules, that is, schedules which, when used as input to algorithm 4.1, cause that algorithm to output "SAFE" and halt. Correctness of the other scheduling algorithms may be proven similarly.

Definition- a schedule block $\prod_i$ of a schedule $\prod$ is **unsafe** if there exists $j,k \in \prod_i, (j \neq k)$ such that either $G_{i-1,j}=c_l$ or $H_{i-1,j}=c_l$, **and** either $G_{i-1,k}=c_l$ or $H_{i-1,k}=c_l$, for some l. A schedule block that is not unsafe is **safe**.

First, we prove two lemmas.

Lemma 1- Algorithm 4.1 outputs SAFE if and only if all blocks of $\prod$ are safe.

Proof:

if - If each block of $\prod_i$ is safe, then at the i^{th} iteration of the algorithm, $\prod_i$ will fail to satisfy the condition of statement 1 in the algorithm. (An unsafe block would satisfy the condition of statement 1, since this is exactly the condition for unsafeness. If it were to satisfy the condition, the algorithm would output UNSAFE and halt.) Since the condition is not satisfied, the algorithm proceeds to compute C_i and examines the next block. After n blocks have been examined, the algorithm outputs SAFE and halts.

only if - The only way that the algorithm can reach the final statement, in which SAFE is output, is if each block $\prod_i$ of $\prod$ is examined and fails to satisfy the unsafety test i step 1. Thus, algorithm 4.1 outputs SAFE only if all blocks in the schedule are safe.

Lemma 2 - Algorithm 4.3 only generates safe schedule blocks.

Proof:

The algorithm takes as input the set S of pairs which are available for scheduling, as well as the current mode C_{i-1} describing the relationship among all the pairs, both those in S and those already scheduled. There are three steps in the algorithm where pairs are scheduled, marked 1, 2, and 3. In step 1, all independent pairs are scheduled. Since, by their definition, they contain no coupled terms, obviously no two of them can share common coupled terms, and thus they cannot cause a block to be unsafe.

In step 2, type-1 and -2 pairs are scheduled. None of them can share coupled terms with the scheduled independent pairs. Nor, since only one pair is selected from any coupling class' type-1 or -2 bins, can they share coupled variables with each other. Scheduling these pairs will not cause the block to become unsafe.

Finally, in step 3, pairs are scheduled from candidate type-3 pairs. The candidate pairs are all chosen from coupling classes which have no type-1 or -2 pairs associated with them. Thus, adding any of these pairs to the pairs already scheduled will not affect the safety of the schedule.

The coupling graph is then formed from these pairs. Edges in the matching are chosen so that none are incident on the same vertex. If two edges do not share a common vertex, then their corresponding pairs do not share a common coupled variable. Thus, by scheduling the pairs corresponding to the matching, we guarantee that none of them share a coupled variable and therefore none cause the block to be unsafe. Thus, algorithm 4.3 only generates safe blocks.

We can now prove the correctness of our scheduling algorithm.

Theorem - Algorithm 4.4 only generates safe schedules, that is, schedules that cause algorithm 4.1 to output SAFE and halt.

Proof:

First, we show that algorithm 4.4 generates schedules. A schedule is a partition $\prod$ of the subterms of the clause head t. S is initially the set of subterms (actually, the set of their indices) in t. *single_step* (described in algorithm 4.2) schedules a subset of S and returns the remainder, that is, after scheduling $\prod_i$ from S, returns a new $S=S-\prod_i$. Algorithm 4.4 continues iterating until S is empty. Since *single_step* always schedules at least one pair from S, algorithm 4.4 will terminate. $\prod$, generated by algorithm 4.4 is a partition of the subterms of t and is therefore a schedule.

Since *single_step* always creates a safe scheduling block (by lemma 2), each block $\prod_i$ in $\prod$ is safe. Algorithm 4.1 halts and outputs SAFE if and only if all blocks of $\prod$ are safe, so it does so on all schedules generated by algorithm 4.4.

4.2. Scheduling with Structures

When adding structures to the scheduling problem, a number of complications appear. First, a finer-grained variety of dependency analysis is needed. SDDA as we have seen it so far provides information on the coupling relationships of subterms *as a whole*. In other words, it might indicate that the second subterm, say, is coupled to the fourth. However, when structures are introduced, it may be the case that the first element of the second subterm is coupled to the first element of the fourth subterm. If other elements are independent, there may be opportunities for additional parallelism. We must design a way to express this finer-grained dependency information in a reasonable notation so that good schedules may be derived.

Secondly, assuming that this notation exists, rules for determining the validity of a schedule must be developed. A number of problems arise here which make this more complicated than when structures are not considered. For example, when can the elements of a structure be unified simultaneously with the unification of the structure's functor, and when must the functor be unified first? Figure 4.4 gives a simple example for each case.

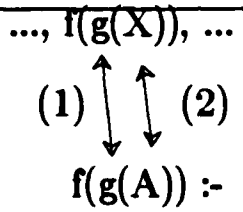


Figure 4.4a) simultaneously unifying functor and element

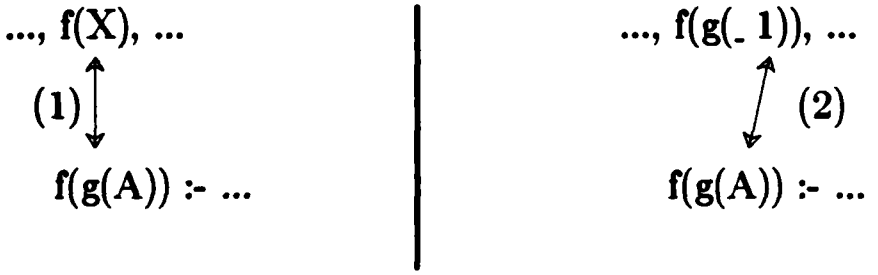


Figure 4.4b) unifying functor first

Another related problem concerns "hidden" structures. A term at compile time may appear textually as a variable, while during the unification it may take on a structure value. It is necessary to take into account these hidden structures as shown in figure 4.5.

It is also necessary to consider special rules for lists, which are themselves a special form of structure.

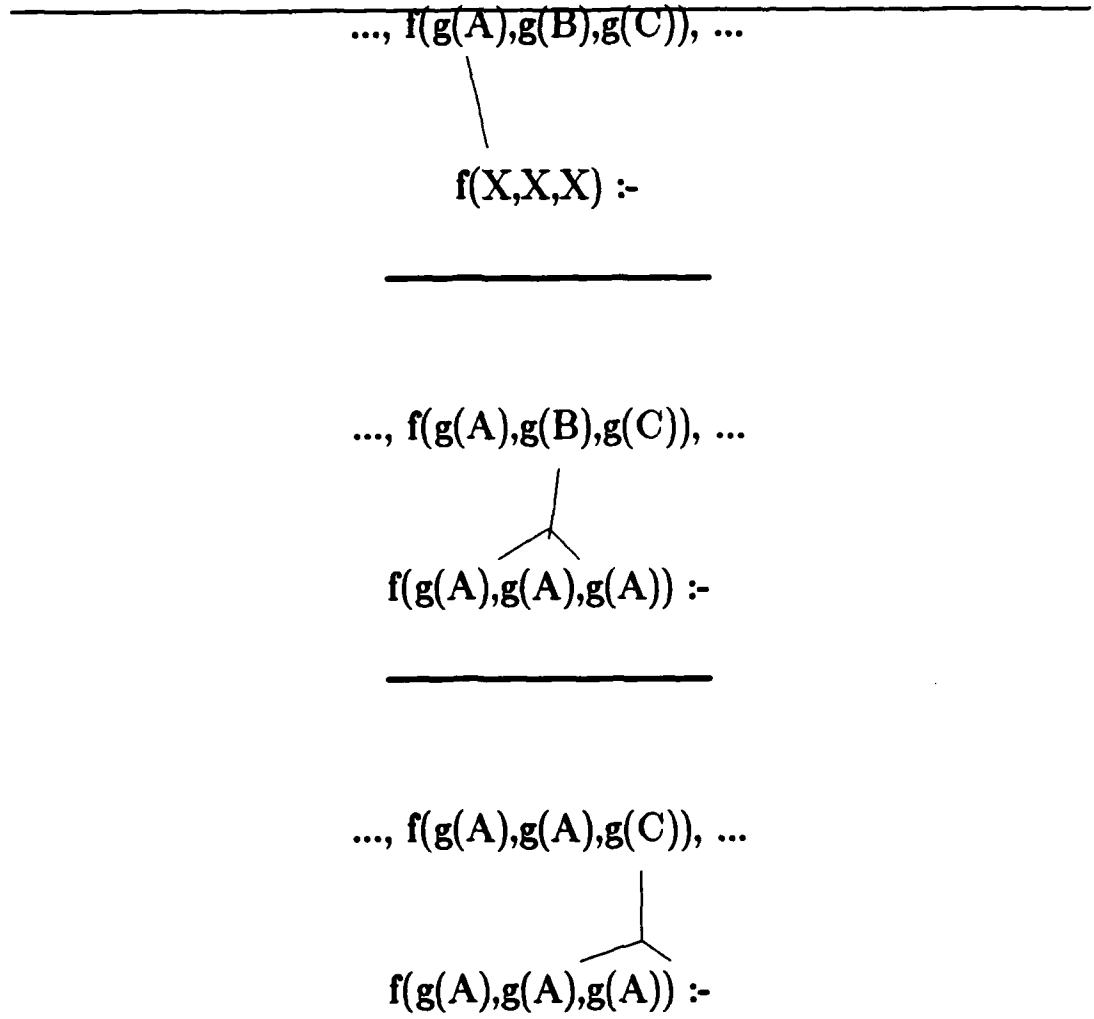


Figure 4.5 - Hidden structure elements and their effect on scheduling

The third complication in the scheduling problem is in the scheduling algorithm. Scheduling involving structures and lists is at least as complex as scheduling without them. New heuristics involving lists and structures must be developed and incorporated into the scheduling algorithm.

The remainder of this chapter is devoted to unification scheduling involving structures and lists. The first part revises the set of definitions presented earlier and describes the expanded mode notation. The second section expands the scheduling rules to include structures and presents a new test for schedule validity, and the third section presents an expended unification scheduling algorithm.

4.2.1. Definitions

The chief change in definitions from those used previously is that of the mode. To schedule unifications when structures are included, modes are

flattened and **labeled**. As an example of flattening, take the term f(X,g(X,Y)). The flattened version of this term is (X,g/2,X,Y). (Remember that the primary functor is unified for free, and therefore does not need to be considered.) By flattening structures and their corresponding modes, all subterms, regardless of depth, become easily accessible for scheduling. In order to distinguish individual subterms (for example, the two X subterms in the above flattened structure), and to reconstruct a term from its flattened version, the subterms are labeled. A label is a tuple indicating the position that the subterm inhabits in the original term. If the first element of the label is n_1, then its corresponding subterm is part of the n_1^{th} subterm of the given term. Likewise, if the second element of the label were n_2, the corresponding subterm would be part of the n_2^{th} subterm of the n_1^{th} subterm of the original term, and so on. For example, the second X would be labeled (2,1). Functor subterms are considered to be the 0^{th} element of a subterm, but for convenience, the final 0 in the label is omitted. The flattened, labeled version of f(X,g(X,Y)) would be ((X,(1)), (g/2,(2)), (X,(2,1)), (Y,(2,2))).

another way to interpret the labels is as the path that must be taken to reach a given subterm in the tree representation of the term, where, if the label is $(l_1, \ldots, l_n)$, the subterm may be reached by first visiting the l_1^{th} son of the root, and then, for each l_i in the label, visiting the l_i^{th} son of the node labeled $(l_1, \ldots, l_{i-1})$. Figure 4.6 demonstrates this for the term f(X,g(X,Y)).

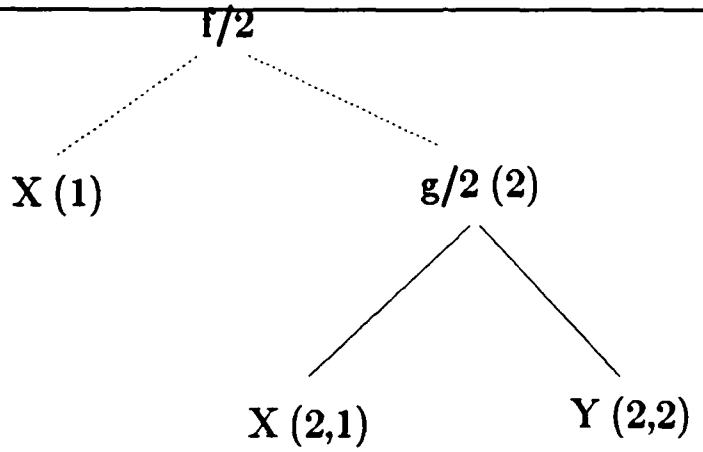


Figure 4.6 - Labeling a term tree

The formal definitions are as follows:

Given a term $t=f(t_1, \ldots, t_n)$, the **flattened term** of t is the tuple $(t_{1,1}, \ldots, t_{1,m}, \ldots, t_{n,1}, \ldots, t_{n,m_n})$ where $(t_{i,1}, \ldots, t_{i,m_i})$ is the flattened subterm of t_i.

Given a **subterm** $t=f(t_1, \ldots, t_n)$, the **flattened subterm** of t is $(f/n, t_{1,1}, \ldots, t_{i,m_1}, \ldots, t_{n,1}, \ldots, t_{n,m_n})$ where $(t_{i,1}, \ldots, t_{i,m_i})$ is the flattened subterm of t_i.

For a constant term $t=f$, its flattened term is the zero-tuple (). For a constant or variable subterm $t=f$ or $t=X$, the flattened subterm is (f) or (X), respectively.

Given a term $t=f(t_1, \ldots, t_n)$ and its flattened term $\hat{t}=(\hat{t}_1, \ldots, \hat{t}_m)$, the **labeled**, flattened subterm of t is $\hat{t}_L=((\hat{t}_1,l_1), \ldots, (\hat{t}_m,l_m))$, where l_i is the label of subterm $\hat{t}_i$. The subterms of two flattened, labeled terms are **corresponding subterms** if they have identical labels. The label l_i of a subterm t_i is determined as follows:

Let the **label prefix** of a term of subterm t (to be explained later) be $(l_1, \ldots, l_m)$. If the *ith* subterm of t is a constant or variable, then its label is $(l_1, \ldots, l_m, i)$. If the *ith* subterm of t, t_i, is a structure $f(t_{i,1}, \ldots, t_{i,n})$, then the label of the functor f/n of t_i is $(l_1, \ldots, l_m, i)$ and the **label prefix** of the subterms $t_{i,1}, \ldots, t_{i,n}$ is $(l_1, \ldots, l_m, i)$.

The label prefix of the main term is the zero-tuple ().

Given a flattened, labeled clause head $\hat{t}_L=((\hat{t}_1,l_1), \ldots, (\hat{t}_k,l_k))$, and a flattened, labeled subgoal, $\hat{t'}_L=((\hat{t}_1',l_1'), \ldots, (\hat{t}_k',l_k'))$ (note that it is now possible that $k \neq n$), the **entry mode** is $C_0=(G_0,H_0)$, where the goal entry mode $G_0=((m_1',l_1'), \ldots, (m_k',l_k'))$. and the head entry mode $H_0=((m_1,l_1), \ldots, (m_k,l_k))$. Each mode element m_i or m_i' may take the value '*i*', denoting a functor of arity r. In section 6.2, we will show how static data-dependency analysis can be improved to yield these entry modes.

Subsequent modes $C_i=(G_i,H_i)$ take similar form, except that the lengths of G_i and H_i may be greater than n and k, respectively, due to the addition of **hidden subterms**. A mode element m_i may be "hidden," in which case it appears as 'g^H', 'i^H', 'c_i^H' or 's^H/r'. A hidden subterm is one that did not appear in the source text of the original pair of terms being unified, but appears later as a result of subsequent unifications. Since it does not appear textually in the original source, it does not have to be scheduled, but its presence may give additional information on the coupling of subterms and will therefore have an influence on scheduling.

4.2.2. Test for schedule validity

The basic principles governing schedule validity are the same as for the case where structures are not included. Two pairs of subterms can be unified if and only if they share no coupled terms. The main differences are that structures must be incorporated into this definition, and the next_C function, which generates a new current mode from the previous current mode and schedule block, must be scheduled to include structures.

A few examples should illustrate the rules concerning structure unification. First, we consider a very simple example with no coupled variables:

t' = f(A,B) (subgoal)
t = f(g(X,Y),Z) (clause head)

The labeled, flattened versions of the terms are:

preceding page was missing from the film

Goal: t' = f(g(X,Y))
Head: t = f(g(A,B))

$\hat{t}_L'$ = ((g/2.(1)), (X,(1,1)), (Y,(1,2)))
$\hat{t}_L$ = ((g/2.(1)), (A,(1,1)), (B,(1,2)))

Again, assume that X and Y are independent at the call. Of course, A and B are independent of each other. The entry modes are:

G_0 = ((s/2,(1)), (i,(1,1)), (i,(1,2)))
H_0 = ((s/2,(1)), (i,(1,1)), (i,(1,2)))

All three elements in the head mode H_0 have corresponding elements in G_0, so all are candidates for parallel unification. In addition, none of the three pairs are coupled to each other and can therefore be scheduled to be unified simultaneously. Thus, the schedule for this unification contains a single block $\prod_1$ = {(1),(1,1),(1,2)} and the final mode C_1=(G_1,H_1) is

G_1 = ((s/2,(1)), (c_1,(1,1)), (c_2,(1,2)))
H_1 = ((s/2,(1)), (c_1,(1,1)), (c_2,(1,2)))

Additional considerations must be made when a structure is unified with a coupled variable. Consider the following terms:

Goal: t' = f(A,A)
Head: t = f(g(X,Y),Z)

and their flattened, labeled versions:

$\hat{t}_L'$ = ((A,(1)), (A,(2)))
$\hat{t}_L$ = ((g/2,(1)), (X,(1,1)), (Y,(1,2)), (Z,(2)))

Assuming that A is unbound at the call, the entry modes are

G_0 = ((c_1,(1)), (c_1,(2)))
H_0 = ((s/2,(1)), (i,(1,1)), (i,(1,2)), (i,(2)))

Subterms labeled (1,1) and (1,2) in the head have no corresponding subterms in the goal and are therefore not candidates for unification at this time. Of the remaining two subterms, only one may be unified because the corresponding terms in the goal are coupled. We choose subterm (1). The resulting mode C_1=(G_1,H_1) is:

G_1 = ((s/2,(1)), (c_2^H,(1,1)), (c_3^H,(1,2)), (s/2,(2)), (c_2^H,(2,1)), (c_3^H,(2,2)))
H_1 = ((s/2,(1)), (i,(1,1)), (i,(1,2)), (i,(2)))

Note that unifying subterm pair (1) caused the two coupled (c_1) mode elements to be replaced with functor and hidden argument mode elements. Since subterm (1) in the goal was coupled to subterm (2), both subterms had to be replaced, and the corresponding arguments were coupled.

At this point, we can either schedule pairs (1,1) and (1,2), or schedule pair (2) alone. We cannot schedule (2) along with (1,1) or (1,2) because subterm (2) in the goal has two hidden arguments, one of which is coupled to subterm (1,1) and one to (1,2). When unifying a functor with a variable, it is necessary to make sure that none of the functor's arguments are coupled with anything else that is being coupled at the same time. (Since hidden arguments are not schedulable, the unification processor unifies all hidden arguments when their non-hidden functor is unified. Section 9 will discuss in more detail.) After unifying pairs (1,1) and (1,2), we get:

G_2 = ((s/2,(1)), (c_2^H,(1,1)), (c_3^H,(1,2)), (s/2,(2)), (c_2^H,(2,1)), (c_3^H,(2,2)))
H_2 = ((s/2,(1)), (c_2^H,(1,1)), (c_3^H,(1,2)), (i,(2)))

We are now free to schedule pair (2):

G_3 = ((s/2,(1)), (c_2^H,(1,1)), (c_3^H,(1,2)), (s/2,(2)), (c_2^H,(2,1)), (c_3^H,(2,2)))
G_3 = ((s/2,(1)), (c_2^H,(1,1)), (c_3^H,(1,2)), (s/2,(2)), (c_2^H,(2,1)), (c_3^H,(2,2)))

We are finished, since subterms (2,1) and (2,2) in the head are hidden and need not be scheduled.

[Although we are only scheduling those subterms which appear textually in the clause head, it may be possible to use hidden structure information to improve unification performance. This will be touched upon in section 9.4.]

The three criteria, then, for schedule safety, are 1) presence of corresponding subterms, 2) independence of simultaneously scheduled pairs, and 3) independence of hidden arguments. All three criteria must be addressed in algorithm 4.5.

Algorithm 4.5 - Extended decision procedure for schedule safety

Input:

A clause head $t=f(t_1, \ldots, t_n)$ and a call subgoal $t'=f(t_1', \ldots, t_n')$.

A flattened, labeled entry mode $C_0=(G_0,H_0)$ containing:

A goal entry mode G_0 computed by SDDA A head entry mode H_0 computed from t.

A schedule $\prod=\{\prod_1, \ldots, \prod_m\}$ for parallel unification of t and t' which is a partition of the set of labels of elements of H_0.

Output:

SAFE if schedule is safe, UNSAFE otherwise.

Algorithm:

for each schedule block $\prod_i$ from $\prod_1$ to $\prod_m$:
  for each $j \in \prod_i$ /* correspondence test */
    if there is no $H_{i-1,j} \in H_{i-1}$ or $G_{i-1,j} \in G_{i-1}$
      [$H_{i,j}$ is the mode element in H_i with label j.
       Similarly for $G_{i,j}$.]
      output 'UNSAFE' and halt.
  for each $j \in \prod_i$
    if there exists a k such that
      $H_{i-1,k} = s^H/m$ (for some m), or
      $H_{i-1,k} = c_m^H$ (for some m), or
      $H_{i-1,k} = i^H$, or
      $H_{i-1,k} = g^H$, or
      $G_{i-1,k} = s^H/m$, c_m^H, i^H, or g^H,
      and j is a prefix of k,
        $\prod_i = \prod_i \cup \{k\}$
  if there exist $j,k \in \prod_i, (j \neq k)$ such that
    at least one of the pairs
    $(G_{i-1,j},G_{i-1,k}),(G_{i-1,j},H_{i-1,k}),(H_{i-1,j},G_{i-1,k}),(H_{i-1,j},H_{i-1,k})$
    is (c_l,c_l) for some l,
      output 'UNSAFE' and halt.
  compute $C_i = next\_C(C_i, \prod_i)$.
    /* note that we compute new mode using hidden arguments,
      as well as scheduled subterms. */

Function $next\_C(C_{i-1}, \prod_i)$
  returns(C_i)

  $(G_i,H_i) = C_{i-1}$.
  for each $j \in \prod_i$
    $(H_i,G_i) = table(H_i,G_i,j)$
  $C_i = (G_i,H_i)$
  return(C_i)

Function $table(H_i,G_i,j)$
  returns(H_i,G_i)

  look up entry $H_{i,j}/G_{i,j}$ in table 4.3
    and modify H_i and G_i according to instructions.
  return new (H_i,G_i)

Table 4.3 extended unification simulation table.

$H_{i-1,j}$ \ $G_{i-1,j}$	g	i	c_k	s/m
g	a	a	b	m
i	a	c	d	h
c_l	e	f	g	j
s/n	n	i	k	l

Note: all entries not mentioned remain unchanged. Also, if an element of H_i or G_i is hidden before being changed, it will remain hidden afterwards.

a) $H_{ij}=G_{ij}=g$.

b) if augmented, replace all c_k in G_i,H_i with g.
if not augmented, $H_{ij}=G_{ij}=g$.

c) $H_{ij}=G_{ij}=c_p$ for some new, unique p.

d) $H_{ij}=G_{ij}=c_k$.

e) if augmented, replace all c_l in G_i,H_i with g.
if not augmented, $H_{ij}=G_{ij}=g$.

f) $H_{ij}=G_{ij}=c_l$.

g) replace all c_k in G_i,H_i with c_l.

h)

$H_{ij}=G_{ij}=s/n$
let $j=(l_1, \ldots, l_p)$
for k = i to n
  add $H_{i,(l_1 \ldots, l_p,k)}=i^H$ to H_i
  $(H_i,G_i)=table(H_i,G_i,(l_1, \ldots, l_p,k))$

i)

$H_{ij}=G_{ij}=s/m$
let $j=(l_1, \ldots, l_p)$
for k = i to m
  add $G_{i,(l_1 \ldots, l_p,k)}=i^H$ to G_i
  if $H_{i,(l_1 \ldots, l_p,k)}$ is hidden
    $(H_i,G_i)=table(H_i,G_i,(l_1, \ldots, l_p,k))$
  /* if not hidden, we schedule them explicitly
    and will have to examine them at the point at which
    they are scheduled. */

j)

for each $k \in G_i$ and H_i such that $k = c_l$
  k = s/m
  let label(k) = $(l_1, \ldots, l_p)$
  for $q = 1$ to m
    if $k \in G_i$
      add $G_{i,(l_1 \ldots, l_p, q)} = c_{p_q}^H$ to G_i
      (where p_q is a new, unique value)
    else add $H_{i,(l_1 \ldots, l_p, q)} = c_{p_q}^H$ to G_i
  let $j = (l_1, \ldots, l_p)$
  for $q = 1$ to m
    $(H_i, G_i) = table(H_i, G_i, (l_1, \ldots, l_p, q))$

k)

for each $z \in G_i$ and H_i such that $z = c_k$
  z = s/n
  let label(z) = $(l_1, \ldots, l_p)$
  for $q = 1$ to n
    if $z \in G_i$
      add $G_{i,(l_1 \ldots, l_p, q)} = c_{p_q}^H$ to G_i
      (where p_q is a new, unique value)
    else add $H_{i,(l_1 \ldots, l_p, q)} = c_{p_q}^H$ to G_i
  let $j = (l_1, \ldots, l_p)$
  for $q = 1$ to n
    if $H_{i,(l_1 \ldots, l_p, q)}$ is hidden
      $(H_i, G_i) = table(H_i, G_i, (l_1, \ldots, l_p, q))$

l)

let $j = (l_1, \ldots, l_p)$
for $k = 1$ to n
  if $H_{i,(l_1 \ldots, l_p, k)}$ is hidden
    then if $G_{i,(l_1 \ldots, l_p, k)}$ does not exist
      add $G_{i,(l_1 \ldots, l_p, k)} = i^H$ to G_i
      /* add a placeholder */
      $(H_i, G_i) = table(H_i, G_i, (l_1, \ldots, l_p, k))$

m)

let $j = (l_1, \ldots, l_p)$
for $k = 1$ to m
  add $G_{i,(l_1 \ldots, l_p, k)} = g^H$ to G_i
  /* expand ground term - all args are ground */
  $(H_i, G_i) = table(H_i, G_i, (l_1, \ldots, l_p, k))$

n)

let $j=(l_1, \ldots, l_p)$
for $k = 1$ to n
  add $H_{i,(l_1,\ldots,l_p,k)}=g^H$ to H_i
  /* expand ground term - all args are ground */
  $(H_i,G_i)=table(H_i,G_i,(l_1, \ldots, l_p,k))$

Note that the table function is recursive in order to simulate unification between (possibly) nested structures. Also, it is important to note that the "atomic" unit of unification scheduling is an element of the flattened version of the clause head. If, in the course of unification, a variable (one of these atomic units) takes on a structure value, the arguments of that structure are not schedulable. They are represented as "hidden" arguments which are implicitly unified by the unification processor which unifies the value associated with the variable and its corresponding subterm. If an explicit structure appears in the clause head, its functor is an atomic unit, as is each element of the structure's flattened subterms. Each of these elements is schedulable. Since the arguments are schedulable, they are not represented by hidden arguments to be explicitly unified when their functor is unified. Rather, the validity testing algorithm waits until they appear explicitly in the schedule before their unification is simulated, just as the unification of these arguments would wait until they explicitly appeared in the schedule.

4.2.3. Scheduling algorithms

In this section, we present the modifications that must be made to the previously presented scheduling algorithms so that they may handle structures.

The modification of algorithm 4.2 is straightforward, and involves a change in the notion of when a partition block is safe. In addition to the conditions previously mentioned, it is only safe to add a the pair p to the block $\prod_j$ if there is an element in G_{j-1} corresponding to p (i.e., if $G_{j-1,p}$ exists) and if, in $\prod_j$, p is a structure with hidden subterms, there is no $q \in \prod_j$, or no r which is a subterm of that q, which is coupled to any of p's hidden subterms. Likewise, when recomputing subsequent modes, if $q \in \prod_i$ for some subsequent block $\prod_i$, there must exist a corresponding $G_{i-1,q}$, and no hidden subterms of elements in $\prod_i$ may be coupled to other elements or hidden subterms of other elements of of $\prod_i$. If this occurs, the block is unsafe and we must backtrack.

In extending the second heuristic algorithm, which attempts to find the maximum maximal matching of the coupling graph, some complications arise. First, in adding edges representing unifications to the coupling graph, only those unifications for which a corresponding element in the current goal mode exists may be considered. Secondly, since structure unifications involving hidden subterms may involve more than two coupling classes, the coupling graph becomes a coupling hypergraph. If p, for example, is a subterm pair to be scheduled in a block $\prod_i$ and $H_{i-1,p}$ = s/n where the arguments of $H_{i-1,p}$ are hidden, and several of them are coupled terms, the unification will interact with all these

coupling classes, so the corresponding "edge" must be incident on all of these coupling class nodes. Thus, we have a hypergraph. Figure 4.7 gives an example of this. Finding the maximum maximal matching of a hypergraph is NP-complete, since it is equivalent to the n-dimensional matching problem [15] , but for the size graphs contemplated there exist relatively efficient exponential backtracking algorithms to find this matching, or equivalently, to find the maximum independent set of the edge graph derived from the hypergraph.

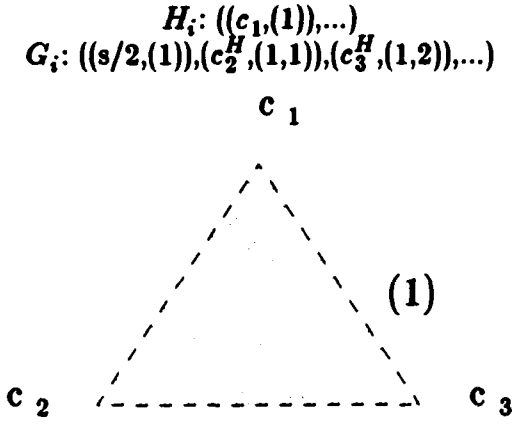


Figure 4.7 - A hyperedge in a coupling hypergraph

In the third heuristic algorithm, we must determine where unifications involving structures fit into our rankings. Table 4.4 shows how these unifications are ranked.

An argument of a structure is **externally coupled** if it is coupled to a term outside the structure.

It should be noted that structure terms are ranked similarly to other unification pairs. The functor itself is considered to be a ground term. If there are two or more coupling classes associated with the unification, it is type-3. If there are none, or none externally coupled, it is independent. If there is exactly one, then the unification is ranked as whatever the argument unification would be ranked if it were not a structure.

As in the second algorithm, type-3 scheduling now involves choosing candidate subterms from those for which corresponding modes exist in the current goal mode. Additionally, the coupling graph is also a hypergraph. As in that case, the graph can be transformed to an edge graph and the largest independent set found.

4.3. Lists

Up to now, it has not been shown how lists fit into the scheme presented here. Lists are a special case of a Prolog structure. As in LISP, the common list notation is shorthand for a more cumbersome car/cdr structure notation. In Prolog, the list functor is ./2, where the first argument is the car of the list and the second argument is the cdr. Thus, [1,2,3] is shorthand to .(1,.(2,.(3,nil))). An

Table 4.4 - Table of unification types

H_{i-1} \ G_{i-1}	c	i	g	s/m
c	3	2	1	3
i	2	i	i	i/2/3
g	1	i	i	i/1/3
s (w/o hidden args)	1	i	i	i
s (w/hidden args)	3	i/2/3	i/1/3	i/w/3

i- independent pair

1- type-1 pair

2- type-2 pair

3- type-3 pair

i/2/3-
If not externally coupled, independent pair. Else if exactly one externally coupled argument, type-2 pair. Else type-3 pair.

i/1/3-
If not externally coupled, independent pair. Else if exactly one externally coupled argument, type-1 pair. Else type-3 pair.

i/1/3-
If not externally coupled, independent pair. Else if exactly one externally coupled argument, then type-2 if coupled argument corresponds to an i argument, otherwise type-1. Else type-3 pair.

alternative is to consider lists to be variable-arity structures, but such an approach would not fit in well with the generality of the scheduling scheme described here, or with static data-dependency analysis. The solution to be used here is to transform all lists into their structure form before SDDA and scheduling are performed. If this is done, no additions need be made to SDDA or the scheduling algorithm.

Some Prolog implementations may have special instructions or data structures designed to increase the efficiency of list handling. In some cases, it may be possible to recover these optimizations by use of implementation-dependent peephole optimizations. An example of this, for the parallel version of the Berkeley PLM, will be presented in chapter 9.

4.4. Complexity of problem

In order to prove that unification scheduling is NP-complete, we will first prove a simple special case that does not satisfy all the requirements of schedule safety to be NP-complete. This special case will be called "simple unification scheduling" and differs from unification scheduling in that unifying two coupling classes does not join the classes, nor does unifying a ground term with a coupled term cause all of the coupled terms to become ground.

DEFINITION:

A schedule $\prod$ is considered **safe for simple unification scheduling** under entry mode $C_0=(G_0,H_0)$ if, for each $\prod_i \in \prod$ then for each $j,k \in \prod_i$ $(j \neq k)$, none of the following are true:

i) $G_{0,j}=c_l$ and $H_{0,k}=c_l$ (for some c_l)

ii) $G_{0,j}=c_l$ and $G_{0,k}=c_l$

iii) $H_{0,j}=c_l$ and $H_{0,k}=c_l$

iv) $H_{0,j}=c_l$ and $G_{0,k}=c_l$

In other words, no distinct unifications in a given schedule step may involve terms in the same coupling class. Note that nothing is said here about joining or grounding coupling classes. Also note that structure subterms are not considered. They will be considered at the end.

DEFINITION:

The simple unification scheduling problem (SUS) is given as follows: Given an entry mode $C_0=(G_0,H_0)$ and a schedule size $D \in Z^+$, is there a schedule $\prod$ with D steps which is safe under unification scheduling?

THEOREM:

SUS is NP-complete.

Proof:

It is simple to show that SUS is in NP. Given an entry mode $C_0=(G_0,H_0)$, create a possible schedule from the elements of the head entry mode. Test this schedule for safety under simple unification. Assuming that the head entry mode has n elements, the worst case would be that in which the schedule had exactly one step with all n elements. Checking each pair would take time $O(n^2)$. The best case would be the schedule in which there were n steps of one element each. Such a check would take linear time. The average case would be a schedule with $\sqrt{n}$ steps of $\sqrt{n}$ elements each. Such a schedule would take time $O(n\sqrt{n})$ to check. In any case, the test of a given schedule may be accomplished in polynomial time.

The completeness part of the proof may be demonstrated through a reduction from resource-constrained scheduling [15] , which can be formulated as follows:

Given m processors, a set T of tasks, each of length l(t) = 1, r resources, resource bounds $B_i = 1$, and resource requirements $R_i(t)$, such that $0 \leq R_i(t) \leq B_i$ for each task t and resource i, and where each task uses no more than 2 resources, and an overall deadline $D \in Z^+$. Is there a m-processor schedule σ for T that meets D and obeys the resource constraints?

We transform the problem as follows:

i) order the tasks (arbitrarily) from $t_1, \ldots, t_T$.

Create a head and goal entry mode H_0 and G_0, respectively, as follows:

ii) For each resource r_j, $1 \leq j \leq r$, create a corresponding coupling class $c(r_j)$ as follows:
if r_j is only used by one task, $c(r_j) = i$.

if r_j is used by more than one task, $c(r_j) = c_j$.

iii) For each task t_i from t_1 to t_T:
if t_i uses no resources,
$G_{0,i}=H_{0,i} = i$.
if t_i uses exactly one resource, r_j,
$G_{0,i} = i$ $H_{0,i} = c(r_j)$
if t_i uses two resources, r_j,r_k,
$G_{0,i} = c(r_j)$ $H_{0,i} = c(r_k)$

Since the use bound on any resource is 1, it is obvious that no two tasks may attempt to use a common resource during the same step of a schedule. A one-to-one correspondence may this be made between scheduled tasks and scheduled unifications, and it is clear that a task schedule σ of length D obeying the resource bounds if and only if a safe simple unification schedule $\prod$ of length D exists.

Thus, SUS is NP-complete.

We now define a more general concept of unification safety that takes into account the joining and grounding of coupling classes.

DEFINITION:
Given a schedule $\prod$ and an entry mode C_0, two coupling classes c_j and c_k are **joined** at schedule step $\prod_i$ if in step $\prod_i$, terms in coupling class c_j or a class joined to c_j at $\prod_i$, and a term in coupling class c_k or a class joined to c_k at $\prod_i$ are unified.

DEFINITION:
Given a schedule $\prod$ and an entry mode C_0, a coupling class c_j is **grounded** at step $\prod_i$ if, in step $\prod_i$, a term in coupling class c_j or some other class which is joined with c_j at $\prod_i$ is unified with either a ground term or a term in a class which is grounded at step $\prod_i$.

DEFINITION:
A schedule $\prod$ is considered **safe for general unification scheduling** under entry mode $C_0=(G_0,H_0)$ if, for each $\prod_i \in \prod$ then for each $j,k \in \prod_i$ $(j \neq k)$, none of the following are true:

i) $G_{0,j}=c_l$ and $H_{0,k}=c_l$ (for some c_l)

ii) $G_{0,j}=c_l$ and $G_{0,k}=c_l$

iii) $H_{0,j}=c_l$ and $H_{0,k}=c_l$

iv) $H_{0,j}=c_l$ and $G_{0,k}=c_l$

unless c_k is grounded one of the r steps immediately prior to $\prod_i$,
AND
for each $\prod_i \in \prod$, for each j,k $\in \prod_i$ $(j \neq k)$ such that $G_{0,j}$ or $H_{0,j}=c_l$, and $G_{0,k}$ or $H_{0,k}=c_m$ (for some $l \neq m$), there is no step $\prod_p$ in the n steps immediately prior to $\prod_i$ such that c_k and c_l are joined in $\prod_p$.

The above definition is the safety criterion expressed in section 4.1.2 when the parameters r and n are arbitrarily large. It is clear that simple unification

scheduling is the special case where r and n are both equal to 0.

DEFINITION:

The general unification scheduling problem (GUS) is given as follows: Given an entry mode $C_0=(G_0,H_0)$, a schedule size D $\in Z^+$, and parameters n and r to the general unification safety definition, is there a schedule $\prod$ with D steps which is safe according to general unification scheduling with parameters n and r?

THEOREM:

GUS is NP-complete.

Proof:

1) It is clear that GUS is in NP, since any possible schedule may be tested for safety in polynomial time using algorithm 4.1.
2) Since SUS is a special case of GUS, GUS is NP-complete.

The above ignores the scheduling of structures. Since unification without structures is a special case of unification with structures, and unification with structures is in NP (using the polynomial safety test of algorithm 4.5), GUS with structures is also NP-complete.

5. Models of Execution

The scheduling scheme described in chapter 4 may be reallized in a number of ways. These may be divided into two broad classes, depending on whether the scheduling operation is performed at compile time or at run time. Each scheme has certain advantages and each provides certain tradeoffs. In addition, special architectural features are required to accommodate the various schemes. Those schemes in which scheduling is performed at compile time are known as static models because scheduling is based on the static source representation of the program and because the schedule, once determined, is not altered at run time. Schemes in which scheduling is performed at run time, when the call takes place, are known as dynamic models, since the schedule is based on the actual values of the variables at the time of the call. Since the values may change from one instance of a call to the next, the unification schedules may differ each time a call is repeated.

5.1. Dynamic Scheduling

In the dynamic scheduling model, the schedule is determined each time a call is performed. Thus, the schedule is based on the actual values of the variables in a call rather than on general predicted values as provided by SDDA. Consequently, there is a greater likelihood of finding an optimal schedule in many cases.

In a typical dynamic scheme, execution of a call involves the following operations:

1) The entry mode of the calling subgoal is computed.
2) The head of the called clause is found and its entry mode is computed.
3) The subterm pairs are scheduled for unification. The scheduling algorithm may be either local or global as described in the previous chapter.
4) The schedule is executed.

There are many optimizations and variations on this general procedure. For example, if execution backtracks to a call and a different clause may be called, execution may be restarted at step 2 above, since the values of the subterms in the calling subgoal will not have changed and the subgoal's entry mode need not be recomputed. The entry mode of the called clause head may be different, however, and must be recomputed.

Since variables in the clause head are always unbound on clause entry and the head entry mode may be determined simply by examining the text of the clause head, head entry modes do not vary from call to call and may be precomputed at compile time and stored for later reference. Thus, step 2, above, may be replaced with an operation which looks up the pre-computed head entry mode of the called clause.

If a local scheduling algorithm is used, the scheduling and unification operations of steps 3 and 4 may be interleaved. In particular, the unification simulation associated with the next_C function may be replaced by the unification itself, after which new modes may be recomputed. In other words, the procedure in figure 5.1a may be replaced by that in figure 5.1b.

a)
<given G_0,H_0>
- schedule Π_1
- compute G_1,H_1 using next_C
- schedule Π_2
- compute G_2,H_2 using next_C
...
- schedule Π_n
- execute Π_1
- execute Π_2
...
- execute Π_n

b)
<given G_0,H_0>
- schedule Π_1
- execute Π_1
- compute G_1,H_1 from current values of t,t'
- schedule Π_2
- execute Π_2
- compute G_2,H_2 from current values of t,t'
...
- schedule Π_n

Figure 5.1
a) non-interleaved scheduling and execution
b) interleaved scheduling and execution

The interleaved execution of figure 5.1b is only an optimization if computing current modes from the actual values of the head and goal is faster than using next_C. The process can be definitely sped up, however, by executing a schedule block in parallel with the computation of the next mode, using next_C, since these operations may be done in parallel. Interleaving the scheduling and unification in this way can reduce the number of scheduling and unification steps by a third.

As an example of a dynamically scheduled unification using the local maximum independent set heuristic, consider the following unification. The calling subgoal is f(A,B,C,D) where B and C have been coupled and A and D are independent. The clause head is f(X,X,Y,Y). Renaming the variables so that coupled variables have the same name and independent variables have distinct names, we have

goal: f(_1,_2,_2,_3)
head: f(_4,_4,_5,_5)

In step 1, the goal entry mode G_0 is computed. It is (i,c_1,c_1,i).

In step 2, the head entry mode H_0 is computed. It is (c_2,c_2,c_3,c_3). As mentioned previously, this can be pre-computed at compile time.

In the interleaved scheduling/unification phase, we must first find a maximum independent set. One such set consists of the first and third subterm pairs. These are scheduled for the first block and immediately executed. In parallel with the unification, the goal and head modes are recomputed. The result is

goal:	f(_1,_2,_2,_3)	goal mode:	(c_2,c_1,c_1,i)
head:	f(_1,_1,_2,_2)	head mode:	(c_2,c_2,c_1,c_1)

Of the remaining unscheduled subterm pairs, either the second or the fourth alone form a maximum independent set. We choose the second, execute it, and recompute the modes and get

goal:	f(_1,_1,_1,_3)	goal mode:	(c_2,c_2,c_2,i)
head:	f(_1,_1,_1,_1)	head mode:	(c_2,c_2,c_2,c_2)

Finally we choose the remaining fourth subterm pair, execute its unification, and are completed.

An architecture which executes such a dynamic scheme needs a number of special hardware features (figure 5.2). First, the architecture needs to be able to compute the entry mode of a call subgoal. Secondly, there needs to be some sort of hardware scheduling unit. In the above case, this unit can choose a maximum independent set, but it may be a unit implementing any scheduling algorithm. In the above example, the hardware needs to be capable of computing new modes from the old modes and the last schedule block, but we have shown that a unit which can compute modes from the terms resulting from execution of a schedule block may be used instead, although one may have to pay a price in efficiency. Finally, the architecture needs to have a number of homogeneous unification units and some method of assigning unification operations to them.

In figure 5.2, the mode memory stores the current modes and the term memory stores the current values of the terms. The mode computer calculates new modes, and the scheduler schedules the subterms for unification. The dispatcher and unification units actually perform the unifications. All of these capabilities may be implemented in microcode or directly in hardware.

The advantage of dynamic scheduling is that precise data dependency information may be used for scheduling. As we shall see, SDDA, used in static scheduling, generates worst-case information. For example, if SDDA indicates that two subterms are coupled, this really means that they may sometimes be coupled. Since scheduling is based on these computed entry modes, the two subterms will not be unified in parallel even if, on some occasion, they happen to be independent. In dynamic scheduling, since we do scheduling from the actual values of subterms, the coupling information is precise. However, we shall see in the next chapter that, in practice, the worst-case information is generally very close to the precise information, since any given Prolog procedure is likely to be called in only a few different ways, and that even where there is a difference

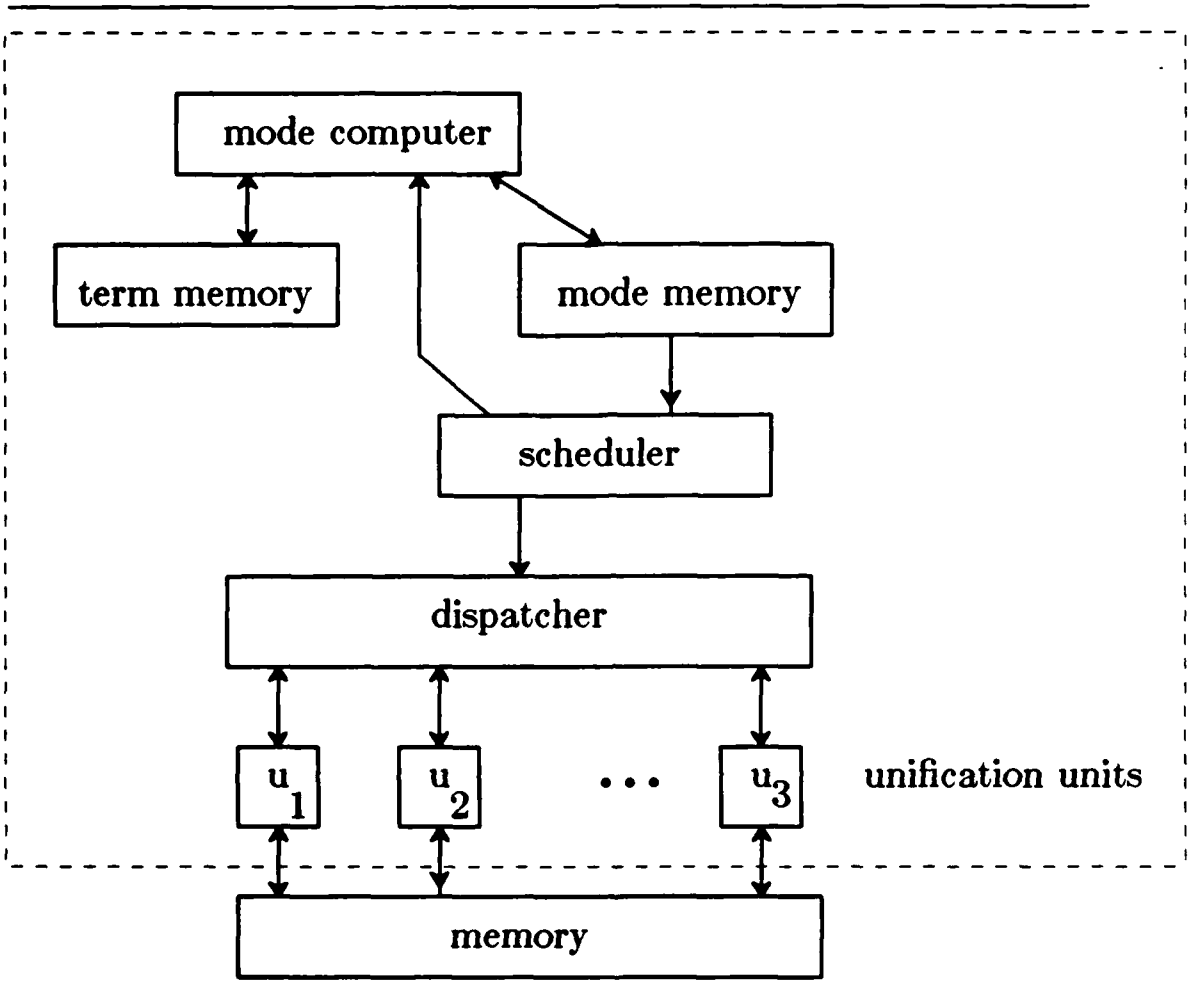


Figure 5.2 - Dynamic scheduling hardware organization

between worst-case and precise information, that difference can be minimized or even eliminated by use of procedure splitting.

In addition to having one not-very-decisive advantage over static scheduling, dynamic scheduling has a number of disadvantages. First is the hardware overhead necessary to implement dynamic scheduling. As shown in the next section, static scheduling requires substantially less hardware. Second is the time overhead needed to compute modes and schedules. Static scheduling also needs to compute these, but it is all done before the program is run, while dynamic scheduling takes time during the actual running of the program. A dynamically scheduled program would run more slowly than the same statically scheduled program, although it would take more time to compile a statically scheduled program. A statically scheduled program would presumably be compiled only once, however.

A third disadvantage of the dynamic model presented here is that the same modes and schedules would have to be computed each time a call is made with the same entry modes. This redundant work can be avoided if modes and their associated schedules are cached. The cache would be an associative memory whose key is the entry mode and which would also contain the schedule derived from that entry mode. Upon executing a call, the entry mode would be computed and looked up in the cache. If the mode is found in the cache, the associated schedule is used. Otherwise the schedule is computed and added to the cache. Such a scheme would speed up dynamic scheduling, but would require a further hardware overhead. Such a cache might have to be quite large.

5.2. Static Scheduling

Chapters 3 and 4 have given the details of static scheduling. Basically, static scheduling involves determination of entry modes and schedules at compile time, and execution of these previously computed schedules at run time. The schedules are included as part of the compiled code. Unlike dynamic scheduling, static scheduling requires a simpler architectural extension (figure 5.3). All that is needed is a set of homogeneous unification units and a dispatcher to assign them unification operations. A synchronization mechanism is needed to insure that unifications in one scheduling block are not started before unifications in the previous one are completed. This may either be a fork/join mechanism or simply having the unification units operate in lockstep. The latter approach will be used here since it appears to be no less powerful than fork/join and requires a simpler control mechanism.

Unlike dynamic scheduling, static scheduling also requires a substantial software system to extract maximum parallelism from a Prolog program's unifications.

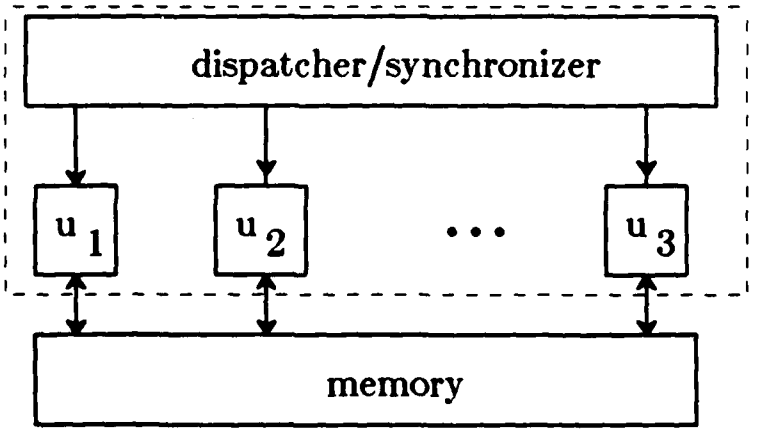


Figure 5.3 - Statically scheduled unification hardware

The disadvantages of the dynamic scheme are the advantages of the static scheme. There is less hardware overhead or scheduling overhead at run time.

Instead, all analysis and scheduling is done once, at compile time, incorporated into the compiled code, and is available for repeated executions of the program without being recomputed.

It appears that static scheduling has a number of advantages over dynamic scheduling. Because of this, we will not consider dynamic scheduling any further, but will instead concentrate on implementing static scheduling.

6. Static Data-Dependency Analysis

Static data-dependency analysis (SDDA) is a pre-compile time technique developed by J-H Chang [2] for determining the coupling relationships between variables and for computing entry and exit modes for clause heads and subgoals. In addition to its use in gathering information for unification scheduling, it is also used for scheduling AND-parallelism and computing intelligent backtracking destinations. Chang provides a description of SDDA in his dissertation. Unfortunately, that description contains a number of errors and omissions, as well as an unrigorous and non-algorithmic presentation of the technique. The next section (6.1) presents a correct and algorithmic description of SDDA and should be considered to supersede the corresponding chapter in [2] . The following section (6.2) presents a number of enhancements to Chang's technique which may be used to improve the data yielded by SDDA and create more efficient unification schedules.

6.1. Description

SDDA is a recursive technique by which the information contained in one or more **query entry modes** (i.e., modes representing the relationships among terms in a top level query) are propagated through a Prolog program so that entry and exit modes for each subgoal and clause are computed. These nodes are identical to those previously described, that is, an n-ary predicate's mode is an n-tuple $(m_1, \ldots, m_n)$ where each m_i is either g (a ground term), i (an independent term), or c_j (a coupled term). (At this point, we are not considering structure modes. They will be considered in section 6.2.)

Intuitively, mode information is propagated by considering in turn each candidate clause of the called procedure. For each subgoal of the clause, an entry mode is computed and the called procedure is similarly examined. When all subgoals in the clause have been examined, an exit mode for the clause is computed which is passed back to the calling subgoal. In order to allow a more efficient analysis and avoid infinite recursion, the notion of "better" and "worse" modes has been created. (These will be defined shortly.) If a procedure is about to be examined, and it has already been examined with a worse entry mode, the new examination is abandoned. Also, a list of all clauses currently "active" (i.e., being examined) is maintained. If a clause is examined which is currently active, and at least one of the activations has an entry mode worse than that of the current clause, the new examination of that clause is abandoned. In this way, infinite recursion is avoided.

When the algorithm is completed, each clause and subgoal in the program which is potentially reachable through the top-level query or queries will possess an entry and exit mode. These represent the relationships between subterms in the predicate (clause head or subgoal) just before and after the predicate is executed, respectively. The framework of the algorithm (6.1) is presented below; the gaps will be filled in later. The reader should assume the existence of two tables maintained by the program and initially empty. One is a table of procedures and their "worst-case" entry and exit modes (i.e., the worst entry and exit modes with which they have been examined). The second is a table of the currently active clauses and their entry modes. Clauses may be identified using the procedure

name and an index. Table management is a simple task in Prolog; entries may be added and deleted using assert and retract, respectively, and lookups are simple Prolog calls.

Algorithm 6.1 - Static Data Dependency Analysis

Input: query entry mode
Output: query exit mode, and entry and exit modes for each reachable clause.

Algorithm:

```
main: query_exit_mode = sdda(query_entry_mode);

sdda(current_entry_mode):
  if worst-case entry mode for the procedure exists /* from table */
   and current entry mode is better than the worst case entry mode,
   and a worst-case exit mode exists for the procedure
  then
    exit mode = worst-case entry mode for that procedure;
    return exit mode;
  else
    worst-case entry mode = owcg(current entry mode,
      worst-case entry mode for procedure);
    /* owcg = optimal worst-case generalization - see below */
    current entry mode = worst-case entry mode;
    replace old worst-case entry mode for procedure in table with new one;

  for each candidate clause in called procedure:
    if current clause is active with an entry mode equal to or
     worse than the current entry mode
    then
      go back to top of loop and try next clause;
    else
      add current clause and entry mode to activation table;
      create variable status V0 from the clause head and current entry mode;
      for each subgoal i from 1 to n  /* n = number of subgoals in clause */
        if subgoal i is unify goal
        then generate Vi from Vi-1 and subgoal;
        else if subgoal i is call
        then
          create a subgoal entry mode for the subgoal using Vi-1;
          subgoal exit mode = sdda(subgoal entry mode);
          create Vi from subgoal exit mode and Vi-1;
        else Vi=Vi-1;  /* for other calls */
      create current exit mode from Vn;
      worst-case exit mode = owcg(previous worst-case exit mode for
        procedure from table, current exit mode);
      replace old worst-case exit mode for procedure in table with new one;
      remove current clause and entry mode from activation table;
  return worst-case exit mode;
```

At this point, there are a number of issues which need to be resolved:

- Defining the worse/better relationship on modes.
- Defining the optimal worst-case generalization.
- Defining the variable status V_i.
- Generating V_0 from the entry mode and the clause head.
- Generating a subgoal entry mode from the subgoal and the previous variable status.
- Generating V_i from a subgoal exit mode and V_{i-1}.
- Generating a clause exit mode from V_n and the clause head.
- Generating V_i from V_{i-1} and a unify goal.

A mode is considered **worse than** another mode if the first mode is more general. In other words, if the first mode will allow creation of a schedule which is safe (although non-optimal) from the second, we say that the first mode is worse than the second.

A number of preliminary definitions must be presented before the better/worse relationship is defined.

DEFINITION:

Let $M=(m_1, \ldots, m_n)$ and $M'=(m_1', \ldots, m_n')$ be two entry modes for the same procedure, and c_j and c_k be mode elements of M and M', respectively, representing coupled terms. c_j **covers** c_k if and only if for each i such that $m_i'=c_k$, $m_i=c_j$.

In other words, c_j covers c_k if $\{i \mid m_i'=c_k\} \subseteq \{i \mid m_i=c_k\}$.

A better/worse partial order may be said to hold on corresponding individual mode elements of M and M' (i.e., m_i and m_i' for any i). (Notation: a > b means "a is worse than b," or "b is better than a," a = b means "a equals b," and a <> b means "there is no relationship between a and b.") The following orderings hold:

i > g	
c_j > g	(for any j)
c_j > i	
$c_j > c_k$	iff c_j covers c_k and c_k does not cover c_j
i = i	
g = g	
$c_j = c_k$	iff c_j covers c_k and c_k covers c_j
$c_j <> c_k$	iff c_j does not cover c_k and c_k does not cover c_j

Using the above relation, we can define a similar better/worse partial order for entire modes:

DEFINITION:

Let $M=(m_1, \ldots, m_n)$ and $M'=(m_1', \ldots, m_n')$ be two modes for the same procedure. Then M is worse than M' ($M>M'$) if and only if there exists an $i, 1 \leq i \leq n$, such that $m_i > m_i'$ and for all other $j, 1 \leq j \leq n$, $j \neq i, m_j > m_j'$ or

$m_i=m_i'$.

Likewise, $M=M'$ if and only if for all i, $1\leq i\leq n$, $m_i=m_i'$. If neither $M>M', M'>M$, nor $M=M'$ holds, then $M<>M'$.

In algorithm 6.1, there are situations in which it is necessary to find a mode which is worse than or equal to both of two other modes. Such a mode is called a worst-case generalization. For example, if a clause is called with two modes M and M', a worst-case generalization M'' can be found from which a schedule can be derived which is safe for both M and M'. We would like M'' to be as "good" as possible so that the maximum potential parallelism will be available in M''. By "as good as possible" we mean that $M''\geq M, M''\geq M'$, and for all M''' such that $M'''\geq M$ and $M'''\geq M'$, it must also hold that $M'''\geq M''$. That is, M'' is the best mode that is worse than or equal to the two modes it is generalizing. We call M'' the optimal worst-case generalization (owcg) of M and M'.

Let $M=(m_1,\ldots,m_n)$ and $M'=(m_1',\ldots,m_n')$ be modes. Then the optimal worst-case generalization $owcg(M,M')=M''=(m_1'',\ldots,m_n'')$ is computed as follows:

for each j from 1 to n
  if m_j = g and m_j' = g, then m_j'' = g
  if m_j = g and m_j' = i, then m_j'' = i
  if m_j = i and m_j' = i, then m_j'' = i
  if $m_j=c_i$ and $m_j'=c_k$, then $m_j''=c_l$ where
    c_l covers both c_i and c_k
  if m_j = i or g and $m_j'=c_k$,
    or $m_j=c_k$ and m_j' = i or g,
    then $m_j''=c_l$ where c_l covers c_k.

Table 6.1 gives some examples of optimal worst-case generalizations.

Table 6.1 - Optimal worst-case generalizations

M	M'	M''
(g,i)	(i,g)	(i,i)
(c_1,c_1)	(g,i)	(c_1,c_1)
(c_1,c_1,i)	(i,c_2,c_2)	(c_3,c_3,c_3)

It can be shown that M'' is the optimal wcg. Let $M''=owcg(M,M')$ be computed by the above algorithm. We examine three cases: $M>M', M=M'$, and $M<>M'$.

1) ($M=M'$) If $M=M'$, then $M''=M$ and $M''=M'$ as follows:
for each $j, 1\leq j\leq n, m_j=m_j'$, which means that

- m_j = g and m_j' = g, in which case m_j'' = g.
- m_j = i and m_j' = i, in which case m_j'' = i.
- $m_j=c_i$ and $m_j'=c_k$, where $c_i=c_k$, in which case $m_j''=c_l$ where $c_l=c_i$ and $c_l=c_k$ (c_l covering both c_i and c_k)

Thus, $M''=M'$ and $M''=M$. For any $M'''\geq M$ or M', it must therefore hold that $M'''\geq M''$.

2) ($M>M'$) Let $M''=owcg(M,Mprime)$ and $M>M'$. Then for each $j,1\leq j\leq n$, one of the following two possibilities holds:
   - $m_j>m_j'$, in which case $m_j''=m_j$.
   - $m_j=m_j'$, in which case $m_j''=m_j$.

   Thus, $M''=M$, and any $M'''\geq M$ must also be $\geq M''$.

3) ($M<>M'$) Let $M''=owcg(M,Mprime)$ and $M<>M'$. Then for each $j,1\leq j\leq n$, one of the following possibilities holds:
   - $m_j>m_j'$, in which case $m_j''=m_j$.
   - $m_j'>m_j$, in which case $m_j''=m_j'$.
   - $m_j=m_j'$, in which case $m_j''=m_j$. (wlog)
   - $m_j<>m_j'$ (only true when $m_j=c_i$, $m_j'=c_k$, and $c_i<>c_k$), in which case $m_j''=c_l$ where c_l covers both c_i and c_k. Thus, $m_j''>m_j$ and $m_j''>m_j'$.

   Thus, by the definition of the $>$ ordering, $M''>M$ and $M''>M'$.

To show that M'' is optimal, let $M'''=(m_1''', \ldots, m_n''')$ be some mode such that $M'''>M, M'''>M'$, and $M''>M'''$. By definition of $>$, there exists a $k,1\leq k\leq n$, such that $m_k''>m_k'''$. Examining the corresponding elements of M and M',

- if $m_k>m_k'$, then $m_k''=m_k$ by definition of owcg. Therefore $m_k>m_k'''$, which implies that $M'''>M$ does not hold; a contradiction.
- If $m_k'>m_k$, then $m_k''=m_k'$. This means that $m_k'>m_k'''$, implying that $M'''>M'$ does not hold; again a contradiction.
- If $m_k=m_k'$, then $m_k''=m_k$ (wlog). This, too, leads to a contradiction.
- If $m_k<>m_k'$, then $m_k''=c_i$ such that c_i covers m_k and m_k'. Assume (wlog) that $m_k'''=c_j$ for some j. Since $m_k''>m_k'''$, c_i must cover c_j, but not vice versa. Since c_i covers m_k and m_k' and c_j does not cover c_i, c_j must not cover either m_k or m_k'. Therefore, either $M'''>M$ or $M'''>M'$ does not hold; again a contradiction.

Therefore, if $M''=owcg(M,M')$, there can be no mode M''' such that $M'''>M$, $M'''>M'$, and $M''>M'''$, and M'' is the optimal worst-case generalization.

Given a clause

$$h :- g_1, \ldots, g_n$$

the **variable status** V_i represents the coupling relationships among the variables occurring in the clause up to and including subgoal g_i. V_i is a triple (G_i,I_i,C_i) where G_i is the set of all variables which are ground after returning from subgoal g_i, I_i is the set of all variables which are independent at that point, and C_i is the partition of all coupled variables into coupling classes. V_0 represents the variable status after the clause head and before the first subgoal.

Variable status triples are constructed from the previous variable status triple (i.e., V_i from V_{i-1}) and the exit mode of subgoal g_i. V_0 may be computed from the clause entry mode and the clause head. The entry mode for a subgoal g_i can be computed from V_{i-1} and the text of the subgoal itself. Finally, the clause exit mode may be computed from the clause head and V_n. The remainder of the section shows how variable status triples and subgoal entry modes are generated.

Execution of a subgoal may be considered to transform the variable status which existed before the subgoal was executed to that which exists after the subgoal is executed. In other words, V_i is a function of a subgoal g_i (the i^{th} subgoal in a clause), its exit mode M_i, and the previous variable status V_{i-1}. The basic principles behind generating a new V_i are the following:

- if two variables in different coupling classes in V_{i-1} are coupled in the exit mode M_i, the coupling classes are joined in V_i.
- if a variable is ground according to M_i, it becomes ground in V_i, regardless of what it was in V_{i-1}.
- a variable is independent in V_i if and only if it is not ground, and either it was independent in V_{i-1} and nothing has been done to change that in M_i (such as binding it to a coupled term), or it is a member of a singleton coupling class.
- all variables in a ground term are ground.
- all variables in an independent term must be assumed to be coupled to each other. We must assume this because it is the worst case. Likewise, we must assume this for coupled terms.

An algorithm for generating V_i is presented as algorithm 6.2.

Generating V_0 from the clause head is a special case of algorithm 6.2. For the input variable status V_{-1}, we use the empty status $G_{-1}=I_{-1}=C_{-1}=\emptyset$, instead of the subgoal, we use the clause head, and instead of the subgoal exit mode, we use the clause entry mode. The algorithm will yield V_0.

Algorithm 6.2 - Generating V_i from V_{i-1}, etc.

Input: Previous variable status $V_{i-1}=(G_{i-1},I_{i-1},C_{i-1})$,
subgoal $t=f(t_1, \ldots, t_n)$, and the subgoal's exit mode $M=(m_1, \ldots, m_n)$.

Output: New variable status V_i.

Algorithm:

```
V_i = V_{i-1};
for each c_j which appears as at least one element in M:
  ccl = ∅;
  for each m_k such that m_k = c_j:
    ccl = ccl ∪ {all variables in t_k}
  for each c_class in C_i:
    if ccl ∩ ≠ ∅
    then
      ccl = ccl ∪ c_class
      C_i = C_i - {c_class}
  C_i = C_i ∪ {ccl}
for each i from 1 to n
  if m_i = g
  then
    G_i = G_i ∪ {all variables in t_i}
    I_i = I_i - G_i
    remove any variables in t_i from all coupling classes in C_i
  else if m_i = i
  then
    ccl = {all variables in t_i}
    for each c_class in C_i
      if ccl ∩ c_class ≠ ∅
      then
        ccl = ccl ∪ c_class
        C_i = C_i - {c_class}
    C_i = C_i ∪ {c_class}
for each c_class in C_i   /* clean up C_i */
  if |c_class| = 1
  then
    I_i = I_i ∪ c_class
    C_i = C_i - {c_class}
  else if c_class ∩ I_i ≠ ∅
  then I_i = I_i - c_class
```

To generate a new V_i when the subgoal is an explicit unify goal A=B, it is possible to transform A=B to a call subgoal =(A,B), where =/2 is a procedure with a single clause =(X,X). This, however, is extra work and will result in

unnecessary worst-case results. Instead, a more direct approach will be used. When A and B are simple variables, table 6.1 indicates the appropriate action to take in transforming V_{i-1} into V_i.

Table 6.2
Unify subgoal transformations on V_{i-1}

A ∈	B ∈ G_{i-1}	B ∈ I_{i-1} or not in V_{i-1}	B ∈ c_class in C_{i-1}
G_{i-1}	a	b	e
I_{i-1} or not in V_{i-1}	c	d	g
c_class in C_{i-1}	f	h	i

First, V_i=V_{i-1}, then

a) No action

b) G_i=G_i ∪ {B}, I_i=I_i - {B}

c) G_i=G_i ∪ {A}, I_i=I_i - {A}

d) Create new coupling class {A,B}, add to C_i. Remove A and/or B from I_i.

e) Add B to G_i, remove B from coupling class in C_i. If coupling class now singleton, move remaining element to I_i. Remove coupling class.

f) Same as e), but use A instead of B.

g) Add A to B's coupling class. Remove A from I_i, if necessary.

h) Same as g), but exchange A and B in above definition.

i) Combine A and B's coupling classes in C_i.

More complex unify goals may be transformed into this above simple one. If A and B are both structures, e.g., f(A_1, . . . ,A_n) = f(B_1, . . . ,B_n), they may be replaced with a series of simpler unifications, e.g., A_1=B_1, . . . ,A_n=B_n.

If only A is a structure, e.g., f(A_1, . . . ,A_n) = B, we follow the following procedure:

- if B is ground, A_1, . . . ,A_n all become ground.
- if A_1, . . . ,A_n are all ground, then B becomes ground.
- Otherwise, replace the above unify goal with f(A_1, . . . ,A_n) = f(B_1, . . . ,B_n) where a coupling class containing B,B_1, . . . ,*Bsbn* is added to V_{i-1}.

To generate a subgoal entry mode, one needs the subgoal (say, the i^{th} one), and the previous variable status triple, V_{i-1}. The algorithm is simple and is given below as algorithm 6.3. Generating a clause exit mode is a special case of the above. Instead of a subgoal, the clause head is used, as is V_n. The result is the clause exit mode.

Algorithm 6.3 - Generating a subgoal entry mode

Input: i_{th} subgoal t = f(t_1, . . . ,t_n),
previous variable status V_{i-1}.
Output: entry mode M = (m_1, . . . ,m_n)

Algorithm:
for j = 1 to n
if t_j contains no independent or coupled variables
then m_j = g
else if t_j contains no coupled variables and any
independent variable in t_j appears in no other
subterm of t
then m_j = i
else if any coupled variables in t_j appear in no other
subterm of t
then m_j = i
/* because the entered clause will only "see" the variable(s)
once and they will seem independent */
else /* linked to other subterms */
if t_j contains a variable which appears in a previous
subterm t_k or contains a variable coupled to a
variable appearing in a previous subterm t_k
(previous: t_k where $1 \leq k < j$)
then $m_j = c_l$ where m_k is c_l
else $m_j = c_l$ for some new, unique, l.

One should note that the clause entry modes described here are equivalent to the goal modes described in chapters 3 and 4, which are used in scheduling. Head modes, as described in chapters 3 and 4, have no real equivalent here, although they may be generated using algorithm 6.3, where $V_n=(G_n,I_n,C_n)$ such that $G_n=C_n = \emptyset$ and I_n = {all variables in the clause head}.*

* One SDDA issue which is irrelevant to unification scheduling, but is important to AND-parallelism and backtracking is the notion of the **generator** of a variable, which is used in generating a dependency graph for the clause. Chang describes the generator of a variable as any previous subgoal in the clause that "contributes to the binding of the variable." Aside from questions about the precise meaning of "contributing to the binding of a variable" (e.g., does adding another variable to X's coupling class "contribute" to X's binding?), the non-enhanced SDDA proposed by Chang cannot handle even simple, obvious cases of "contributing to binding." For example, in the subgoal
..., Z = g(X,Y), f(Z), ...
where X, Y, and Z are all independent before the call, the entry mode of the call will simply be entry(f,1,(i)), since structure elements are not distinguished. If f contains the single clause
f(g(a,_)).
X will have been bound, but the exit mode of the call will still be exit(f,1,(i)), and Z will still be independent since Y is independent. Also, there is no evidence of X having been

References

1. J.-H. Chang and A. M. Despain, "Semi-Intelligent Backtracking of Prolog Based on A Static Data Dependency Analysis," *Proceedings of the Symposium on Logic Programming - 1985*, Boston, July 1985.
2. J.-H. Chang, "High Performance Execution of Prolog Programs Based on a Static Data Dependency Analysis," Ph.D. Thesis, University of California, Berkeley, October 1985. Available as Tech. Report UCB/CSD 86/263
3. J.-H. Chang, A. M. Despain, and D. DeGroot, "AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis," *Proceedings of Spring 1985 Compcon*, San Francisco, March 1985.
4. Wayne Citrin, Peter Van Roy, and Alvin M. Despain, "Compiling Prolog for the Berkeley PLM," *Proceedings of the Hawaii International Conference on Systems Sciences - 1986*, Honolulu, HI, January 1986.
5. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
6. John S. Conery, "The AND/OR Model for Parallel Interpretation of Logic Programs," Ph.D. Thesis, Dept. Information and Computer Science, Univ. Calif., Irvine, 1983.
7. S. A. Cook, "An Overview of Computational Complexity," *Comm. of the ACM*, vol. 26, no. 6, pp. 401-407, June 1983.
8. A. M. Despain, W. Kahan, R. M. Karp, E. L. Lawler, Y. N. Patt, and A. J. Smith, *A Research Proposal to Investigate a High Performance Multiprocessor, Logic-Machine Architecture*, 2, January 23, 1985. NSF research proposal
9. Tep Dobry, "A Prolog Machine Architecture," Technical Note, Computer Science Division, UCB, July 1984.
10. T. P. Dobry, Jung-Herng Chang, Alvin M. Despain, and Yale N. Patt, "Extending a Prolog Machine for Parallel Execution," *Proceedings of the Hawaii International Conference on Systems Sciences - 1986*, Honolulu, HI, January 1986.
11. C. Dwork, P.C. Kanellakis, and J. C. Mitchell, "On the Sequential Nature of Unification," *The Journal of Logic Programming*, vol. 1, no. 1, pp. 35-50, June 1984.
12. B. Fagin. Personal communication
13. K. Furukawa and T. Yokoi, "Basic Software Syatem," *Procedings of International Conference on Fifth Generation Computers*, Tokyo, Japan, November 6-9, 1984.
14. H. Gallaire and C. Laserre, "Metalevel Control for Logic Programs," in *Logic Programming*, ed. S. A. Tarnlund, pp. 173-185, Academic Press, London,

bound. The call to f is certainly a generator of X and Z, but there is no way to deduce that here. A solution to the problem will not be proposed here except to suggest that the enhancements discussed in section 6.2 may be relevant to a solution.

1982.

15. M. R. Garey and D. S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Francisco, 1979.
16. N. Ito, H. Shimizu, M. Kishi, E. Kuno, and K. Rokusawa, "Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog," *New Generation Computing*, vol. 3, pp. 15-41, 1985.
17. D. E. Knuth and P. Bendix, "Simple Word Problems in Universal Algebras," in *Computational Problems in Abstract Algebra*, ed. J. Leech, pp. 263-297, Pergamon, London, 1970.
18. R. A. Kowalski, *Logic for Problem Solving,* North-Holland/Elsevier, New York, 1979.
19. J. Maluszynski and H. J. Komorowski, "Unification-free Execution of Logic Programs," *Procedings of the 1985 Symposium on Logic Programming*, pp. 78-86, Boston, July 15-18, 1985.
20. C. S. Mellish, "The Automatic Generation of Mode Declarations for Prolog Programs," DAI Research Paper 163, Dept. of Artificial Intelligence, Univ. of Edinburgh, August 1981.
21. C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity,* Prentice-Hall, Englewood Cliffs, NJ, 1982.
22. M. S. Paterson and M. Wegman, "Linear Unification," *Journal of Computer and System Sciences*, vol. 16, pp. 158-167, 1978.
23. L. Pereira and A. Porto, "Selective Backtracking," in *Logic Programming*, ed. S. A. Tarnlund, pp. 107-114, Academic Press, London, 1982.
24. C. G. Ponder and Y. N. Patt, "Alternative Proposals for Implementing Prolog Concurrently and Implications Regarding their Respective Microarchitectures," *Proceedings of the 17th Annual Microprogramming Workshop*, New Orleans, November 1984.
25. P. Van Roy, "A Prolog Compiler for the PLM," Masters' Report, University of California, Berkeley, August 1984.
26. D. H. D. Warren, "Applied Logic - Its Use and Implementation as Programming Tool," Ph.D. Thesis, Univ. Edinburgh, Scotland, 1977. Available as Tech. Note 290, AI Center, SRI International
27. N. S. Woo, "A Hardware Unification Unit: Design and Analysis," *Proceedings of the 12th Intl. Symposium on Computer Architecture*, New Orleans, June 1985.
28. H. Yasuura, "On Parallel Computational Complexity of Unification," *Proceedings of International Conference on Fifth Generation Computers*, Tokyo, Japan, November 6-9, 1984.